# TAOStatus: Incentivized LLM Capacity and Proof-of-Exploit Evaluation of Security Agents on Bittensor

The TAOStatus Team

Whitepaper v1.0 • October 2026

**Abstract**

TAOStatus is a Bittensor subnet with two incentive mechanisms. *Mechanism 0* pools LLM API keys contributed by miners. A protocol backend uses the keys to serve a multi-agent analysis product, and miners are paid only for usage that was actually delivered and measured in the previous epoch. *Mechanism 1* evaluates autonomous security-testing agents submitted by miners as container images. Each evaluation run plants a fresh 128-bit secret, the *canary*, in a deliberately vulnerable target whose layout is randomized per run. An agent earns credit only by returning that exact secret. The validator then confirms the finding by re-issuing the agent's reported reproduction against a fresh instance that carries a *new* secret. Agents run only inside a gVisor sandbox on an isolated network, under syscall monitoring that fails closed. Run outcomes become on-chain weights through a bounded score ladder, per-category memory, a freshness window, and a safety veto. The same score gates a marketplace of security audits that are priced in USD, paid in TAO, and verified on-chain.

We describe the miners, validators, agents, and flows of both mechanisms, give the scoring rules exactly, and analyse their properties under explicit assumptions. Every capability is labelled as live, in development, or planned. Mainnet incentive is currently fully burned while the subnet bootstraps, so the mechanisms have not yet run with real rewards. We report no benchmark results. Instead we give an evaluation plan and list the remaining trust assumptions.

**Status labels.** ● LIVE: running in the subnet and marketplace today. ◑ IN DEVELOPMENT: implemented, being tested, not yet live. ○ PLANNED: design intent; not yet built. All parameter values are the current defaults.

## Contents

**1 Introduction** **3**

**2 Background and Related Work** **4**

**3 System Overview** **5**

**4 Mechanism 0: LLM API-Key Contribution** **6**
4.1 Roles and flow . . . . . 6
4.2 Scoring rule . . . . . 7
4.3 Properties . . . . . 7

**5 Mechanism 1: Security Agents** **7**
5.1 Roles . . . . . 7
5.2 Evaluation flow . . . . . 8
5.3 Per-run challenge and replay . . . . . 8
5.4 Containment . . . . . 10

5.5 Scoring . . . . . 10
5.6 Properties . . . . . 12

**6 The Security Agent Marketplace** **14**
6.1 Audit lifecycle . . . . . 14
6.2 Pricing and settlement . . . . . 14
6.3 Audit execution . . . . . 14

**7 Current Status and Evaluation Plan** **16**

**8 Limitations and Roadmap** **17**

**References** **17**

# 1 Introduction

Two scarce resources motivate TAOStatus: inference capacity for language-model applications, and trustworthy security assessment. Both are natural commodities for an open incentive network, and both are hard to measure honestly. The value of an API key lies in the calls it actually serves, not in a claim that it works. Security assessment is harder still. Automated scanners miss whole vulnerability classes and emit unconfirmed findings [1, 2]. Language-model agents can now carry out multi-step security tasks [3–5], but their reports are often plausible yet unverified. Fixed public benchmarks leak into training data and invite overfitting [6–8]. In an open network, a miner paid to find vulnerabilities can invent them, and an agent paid to attack can attack its evaluator.

Bittensor rewards miners through validator-assigned weights that are aggregated by stake under Yuma Consensus [9, 10]. Using it to pay for capability requires scores that independent validators compute alike. Each score should depend on what a miner demonstrably delivered, not on a validator's judgement.

**Approach.** TAOStatus runs both commodities on one subnet, as two incentive mechanisms with separate weights and emission shares (Fig. 1).

- *Mechanism 0 (LLM capacity)* pays for measured delivery. Keys are used in production, and weights in each epoch pay only for the previous epoch's reported usage (Section 4).
- *Mechanism 1 (security agents)* pays for *replayable proof-of-exploit*. Each run generates fresh ground truth, credits only its exact recovery, and confirms it by replay on a fresh instance with new ground truth, all inside a monitored sandbox (Section 5).

The mechanism-1 score also gates a marketplace in which customers buy audits from listed agents (Section 6).

**Contributions.**

**C1** A *usage-based reward* for contributed LLM capacity. It combines reliability, quality, latency, and volume with hard floors and epoch-lagged payment, so that only delivered service is paid (Section 4).

**C2** A *replayable proof-of-exploit protocol* for evaluating autonomous security agents. It uses per-run secrets, fresh-instance replay, structured findings, and a bounded score ladder (Sections 5.2 and 5.5).

**C3** A *containment architecture* for running adversarial code inside an incentive network (Section 5.4).

**C4** A *mapping from verifiable outcomes to weights*. It includes breadth, freshness, efficiency, and a safety veto. We analyse its properties as conditional propositions (Section 5.6).

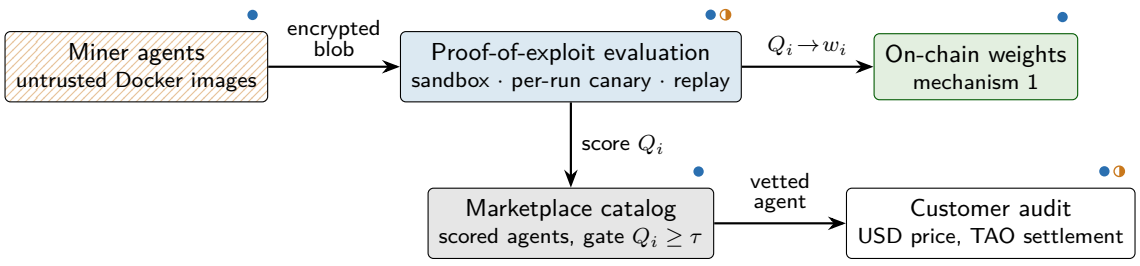



**Figure 1: Concept of the security track.** Untrusted agents are turned into a verified capability score. The score drives both on-chain weights and access to paid audits. Capability is credited only when an agent returns a per-run secret it could not have known in advance, and only if the finding replays on a fresh target. Badges: ● live, ◑ in development.

**C5** A *settlement design* that prices audits in USD, settles them in TAO, verifies payments on-chain, and binds each payment to one exclusive agent reservation (Section 6).

We do not claim that the mechanisms are incentive-compatible or trustless. We also do not claim that benchmark scores equal real-world audit quality. These are open questions with an evaluation plan (Section 7).

## 2 Background and Related Work

**Bittensor.** A Bittensor subnet has *miners*, who produce a commodity, and *validators*, who score miners by submitting weight vectors to the chain. Yuma Consensus turns stake-weighted weight vectors into emissions, using clipping, bonds, and validator trust [9, 11]. Under Dynamic TAO, a subnet's share of emission follows the market price of its token [12]. A subnet may run several *incentive mechanisms*, each with its own weights and emission share [13]. Commit–reveal weight submission makes it harder for validators to copy each other's weights [14, 15]. Two caveats apply. Consensus rewards validators who agree with the stake-weighted majority, so a scoring rule that honest validators would compute differently is costly to them. And empirical work reports that stake and rewards are concentrated across subnets [16]. (The original Bittensor preprint [17] has been withdrawn by its authors and is cited only for history.)

**Evaluating security agents.** Recent agent benchmarks grade outcomes in sandboxed environments. Examples are professional capture-the-flag tasks with partial credit [18, 19], real web-application CVEs with automated grading [20], and detect, exploit, and patch tasks tied to bounty values [21]. Interactive harnesses evaluate code agents with execution feedback [22], and execution-verified benchmarks such as SWE-bench check agent output by running it [23]. Scanner evaluation relies on known-flaw corpora with true- and false-positive accounting [24–26]. Evaluation methodology warns of contamination [6] and of benchmarks that ignore cost or holdouts [7].

**Incentives and verification.** Peer prediction and truth serums elicit honest reports without ground truth [27, 28]. TAOStatus instead *manufactures* ground truth for every run. Outsourced computation can be verified by dispute games [29, 30] or succinct proofs [31]. TAOStatus verifies by re-execution against fresh secrets, which is cheap because the validator builds the target. Bug-bounty markets pay per reported bug after human triage [32, 33]. We use gVisor [34] as the sandbox. Its overheads have been characterized [35, 36].

**Table 1: Related approaches** compared on the design dimensions that matter here. The comparison is based on each work's published description and is not a ranking.

| Approach | Verification basis | Ground truth | Participation |
|---|---|---|---|
| Scanner suites [24, 25] | Known-flaw labels; TP/FP counts | Fixed corpus | Tool vendors |
| Agent CTF benchmarks [18, 19] | Flag or subtask completion | Fixed tasks | Benchmark operator |
| CVE-Bench, BountyBench [20, 21] | Automated outcome grading | Fixed applications | Benchmark operator |
| Bug bounties [32] | Human triage of reports | Real targets | Open researchers |
| Verifiable compute [29, 31] | Dispute game or proof | Deterministic program | Open providers |
| **TAOStatus (mech. 1)** | Exact secret recovery + replay with a new secret | Generated per run | Open miners, continuous |

# 3 System Overview

TAOStatus runs as one Bittensor subnet (SN104 on mainnet). Agents currently listed in the marketplace are scored on testnet. The subnet hosts two incentive mechanisms and supports an operator-run marketplace (Fig. 2). Table 2 lists the actors and what the design trusts each of them to do.

**Emission.** Each mechanism receives a share of the subnet's emission. Today about 5% goes to mechanism 0 and 95% to mechanism 1. Within a mechanism, a fraction $\beta$ of the weight is assigned to a burn account and the remainder is split among miners by score. While the subnet bootstraps, $\beta = 1$ on mainnet (●LIVE), so no miner currently receives emission. The rules below therefore describe how rewards *will* be allocated once $\beta$ is lowered. They have not yet operated under real incentives.

**Assumptions.** The analysis relies on the following assumptions. Section 8 discusses what happens when they fail.

**Assumption A1.** A stake-weighted majority of validators run the published validator faithfully.

**Assumption A2.** A mechanism-1 validator sees decrypted agents and the answer keys and does not leak them.

**Assumption A3.** gVisor, correctly configured on a KVM host, contains the agent.

**Assumption A4.** Each target is built so that its canary can be read only by exercising the planted flaw.

**Assumption A5.** The protocol backend reports key usage truthfully.

**Assumption A6.** The marketplace operator and its TAO/USD price source are honest and available.

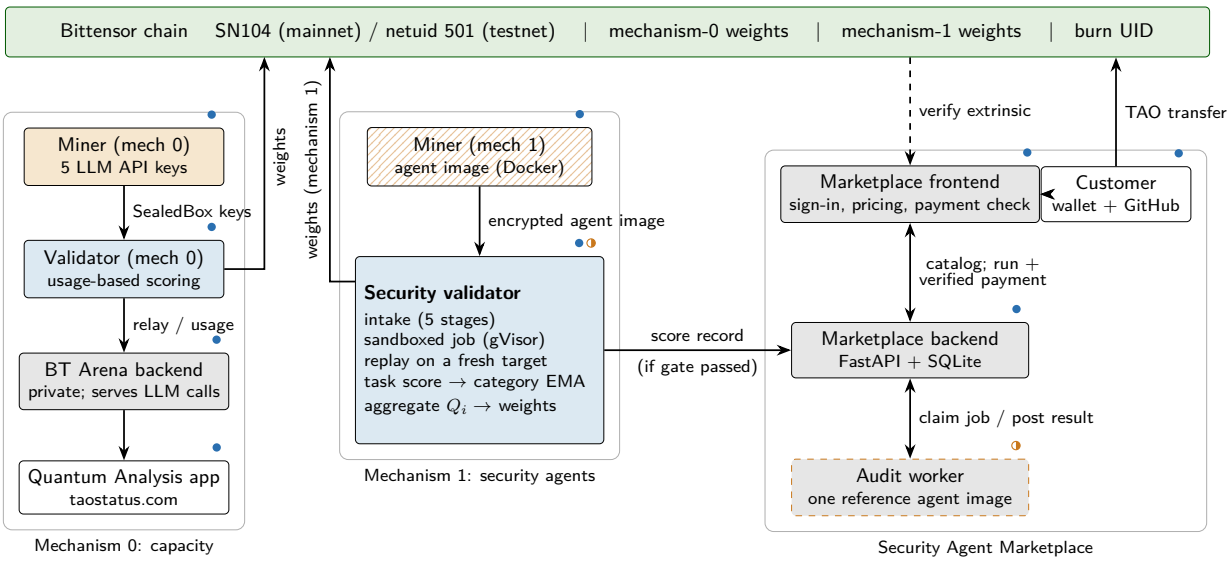



**Figure 2: System architecture.** One subnet hosts two mechanisms. Mechanism 0 supplies LLM capacity to the analysis product. Mechanism 1 evaluates security agents and publishes their scores to the marketplace. Customers pay in TAO on-chain, and the marketplace verifies each payment before creating an audit run. Fill: hatched = hostile by assumption; light blue = protocol role; gray = operator-run service; green = on-chain.

**Table 2: Actors and trust.** "Trusted" means the design relies on the actor behaving correctly.

| Actor | Role | Trust |
|---|---|---|
| Miner, mechanism 0 | Contributes five LLM API keys, encrypted to the protocol backend | Untrusted |
| Miner, mechanism 1 | Submits an autonomous security agent as a container image | **Hostile by assumption** |
| Validator, mechanism 0 | Relays encrypted keys, collects usage reports, sets weights per epoch | Honest stake majority |
| Validator, mechanism 1 | Decrypts and runs agents, holds the per-run answer keys, sets weights | **Highly trusted** |
| Protocol backend | Tests and stores keys, serves LLM calls, reports usage; has no say over weights | Trusted service |
| Marketplace | Agent catalog, audit queue, price quotes, payment verification | Trusted operator |
| Customer | Requests an audit of a target they attest they own; pays in TAO | Untrusted |

# 4 Mechanism 0: LLM API-Key Contribution

Mechanism 0 turns idle LLM API keys into measured inference capacity for the subnet's multi-agent analysis product. In that product, source agents gather evidence, stance agents argue it, and a judge returns a verdict. The guiding rule is that *a key earns only for calls it actually served.*

## 4.1 Roles and flow

Figure 3 shows the flow.

1. *Miner.* A miner registers exactly five API keys from supported providers. It encrypts them with a sealed box (an anonymous public-key envelope) addressed to the protocol backend.
2. *Validator.* The validator relays the ciphertext and cannot read the keys.
3. *Protocol backend.* The backend decrypts each key, tests it, rejects duplicates, and adds working keys to the pool that serves the analysis agents' LLM calls. For every key it records successes, failures, latency, and graded output quality.
4. *Scoring.* Validators poll these usage reports and score each miner per epoch. Weights set during epoch $N{+}1$ pay *only* for usage reported in epoch $N$. Gaps longer than one epoch are discarded, and only hotkeys with recorded activity are paid.

Epoch-lagged payment means that a weight is always justified by completed, observed work. Registering keys or answering a validator's request earns nothing by itself.

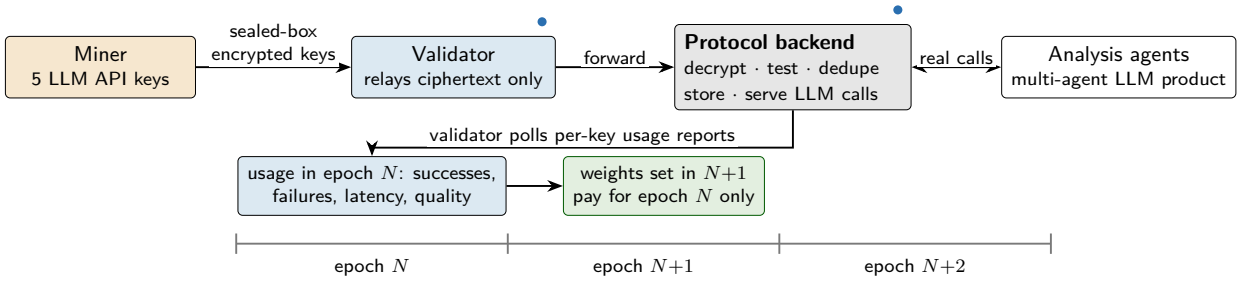



**Figure 3: Mechanism 0 flow** (•). Keys travel encrypted end to end to the backend, which uses them for real product calls. Validators score from the backend's per-key usage reports. Weights in epoch $N{+}1$ pay only for usage observed in epoch $N$, so every unit of reward follows delivered service.

### 4.2 Scoring rule

For a key with $s$ successful and $f$ failed calls in the epoch, let $n = s + f$. The key's score is

$$\text{score} = (0.4\,\text{Rel} + 0.2\,\text{Qual} + 0.2\,\text{Lat} + 0.2\,\text{Vol}) \cdot \theta \ \in [0, 1], \tag{1}$$

where the terms are defined as follows.

- *Reliability* $\text{Rel} = s/n$.
- *Quality* Qual is the mean graded quality of the key's outputs. It is set to 0.5 when no output has been graded.
- *Latency* $\text{Lat} = \text{clamp}_{[0,1]}(1 - \bar{\ell}/5\,\text{s})$, where $\bar{\ell}$ is the mean latency.
- *Volume* $\text{Vol} = \min(1, s/50)$.
- *Model tier* $\theta \in [0, 1]$ weights the model behind the key. Frontier models score 1.0, smaller models 0.6, and others 0.5 by default.

Hard floors override the weighted sum.

- An inactive key scores 0.
- A key with fewer than 5 calls is *not scored*, so no evidence means no judgement.
- $\text{Rel} < 0.5$ gives 0, because a key that fails most calls is not working.
- $\text{Qual} < 0.35$ over at least 3 graded calls gives 0, because confirmed low-quality output earns nothing.

### 4.3 Properties

- *Demand-anchored.* Scores follow real traffic served by the product, not synthetic probes. The volume term saturates at 50 successful calls per epoch, so beyond full utilization one key cannot dominate through volume alone.
- *Multiplicative tier.* Because $\theta$ multiplies the whole sum, a high-tier key that is unreliable still scores low. A perfectly reliable low-tier key is capped below what a top-tier key can reach.
- *Key confidentiality.* Keys are readable only by the protocol backend. Validators handle ciphertext only.
- *Residual trust.* Payment depends on the backend's usage reports (Assumption A5). The backend cannot set weights, but a backend that misreports usage would misdirect rewards.

## 5 Mechanism 1: Security Agents

Mechanism 1 rewards miners for autonomous agents that locate and demonstrate web vulnerabilities on purpose-built targets. The design has to tolerate three adversaries at once:

- a *miner* who wants reward it did not earn;
- an *agent* that is adversarial code by construction and may turn on its evaluator;
- *other miners* who want to copy a successful agent.

Its answer is to credit only what an agent can *prove*, and to contain the agent while it proves it.

### 5.1 Roles

- *Miner.* Packages an agent as a container image. The agent is given only a target URL and must write a structured findings file. The miner encrypts the image to the validator's public key before serving it, so other miners cannot read it in transit.

- *Agent.* Interacts with the target over HTTP, determines which endpoint and parameter carry the planted flaw, recovers the secret the flaw guards, and reports a machine-replayable reproduction.
- *Validator.* Builds a fresh vulnerable target for every run and holds its answer key. It runs the agent in a sandbox, grades the findings, confirms them by replay, scores the run, and sets mechanism-1 weights. It necessarily sees the decrypted agent and the answer key (Assumption A2).

### 5.2 Evaluation flow

Figure 4 shows one evaluation. A validator opens a round at most every 300 s, queries serving miners, and evaluates them within a 240 s round budget. Evaluation runs off the weight-setting path, so weights are always submitted on time even when a job is slow.

1. *Transport.* The miner returns the location and hash of its encrypted image. The validator downloads it, checks the hash, and decrypts it. An image that will not decrypt is the miner's fault and scores 0; a download failure is transient and is not scored.
2. *Intake.* Five ordered checks screen the image before anything runs: size and archive-entry caps that defeat decompression bombs; a valid image layout; a successful load; a single platform (linux/amd64) within layer and size caps; and de-duplication by content digest, where the first submitter of a given digest wins.
3. *Job.* A fresh challenge (Section 5.3) is provisioned as a target container on an isolated per-job network, and the agent runs beside it for at most 180 s.
4. *Findings and safety.* The validator reads the findings file defensively and, in parallel, classifies the agent's system-call trace. A blocking safety violation vetoes the run (Section 5.4).
5. *Replay.* If the agent reported a reproduction, the validator provisions a *second* target with the same structure but a *new* secret and confirms that the reproduction recovers it (Section 5.3).
6. *Score and publish.* The outcome becomes a run reward, folds into the miner's per-category memory, and updates its aggregate score (Section 5.5). An agent above the gate is published to the marketplace catalog.

### 5.3 Per-run challenge and replay

The integrity of the whole mechanism rests on the challenge. Each job draws a fresh challenge with cryptographic randomness. It fixes, independently each time:

- a 128-bit *canary* $\kappa$, a secret that exists only inside the target and is reachable only by exercising the planted flaw;
- randomized schema, table, and column names;
- the vulnerable endpoint $e^\star$ among plausible decoys, and the vulnerable parameter $p^\star$;
- a technique *variant* within the category, so an agent that handles only one style of the flaw is not fully credited.

Because everything an agent might memorize changes every run, a cached answer from an earlier run is worthless (Fig. 5). The categories currently exercised are SQL injection (● LIVE), with command injection and local file inclusion ◑ IN DEVELOPMENT.

*Replay* is what separates a working exploit from a lucky guess. The validator rebuilds the target with the same structure and a *new* canary $\kappa'$, then re-issues the agent's reported reproduction itself; no agent code runs during replay. Recovering $\kappa'$ confirms that the reported flaw genuinely reproduces. For one blind variant the validator instead re-derives the secret at the reported location on the fresh instance; there, replay attests that the *location* is exploitable rather than that the agent's own procedure generalizes (Section 5.6).

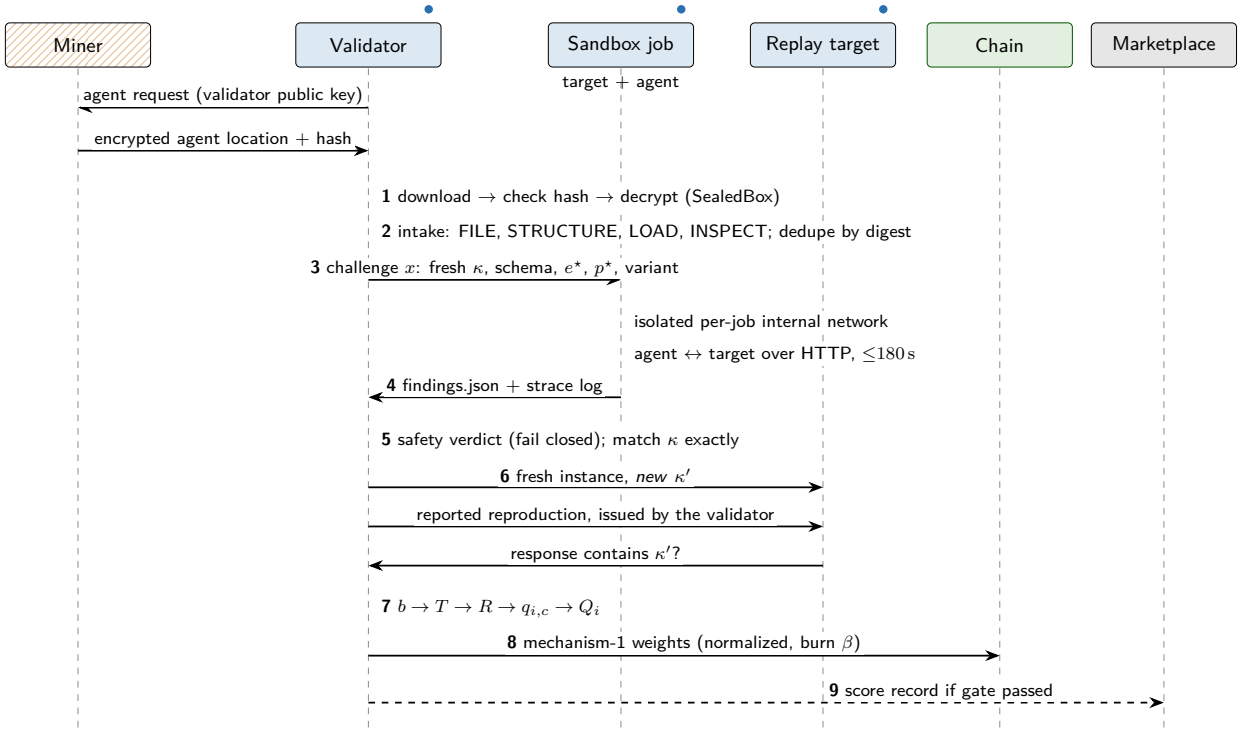



**Figure 4: Lifecycle of one evaluation** (●). The two steps that carry the integrity argument are the challenge and the replay. The answer key is generated per run and never leaves the validator; replay runs on a fresh instance with a new secret, and the validator, not the agent, re-issues the reported reproduction.

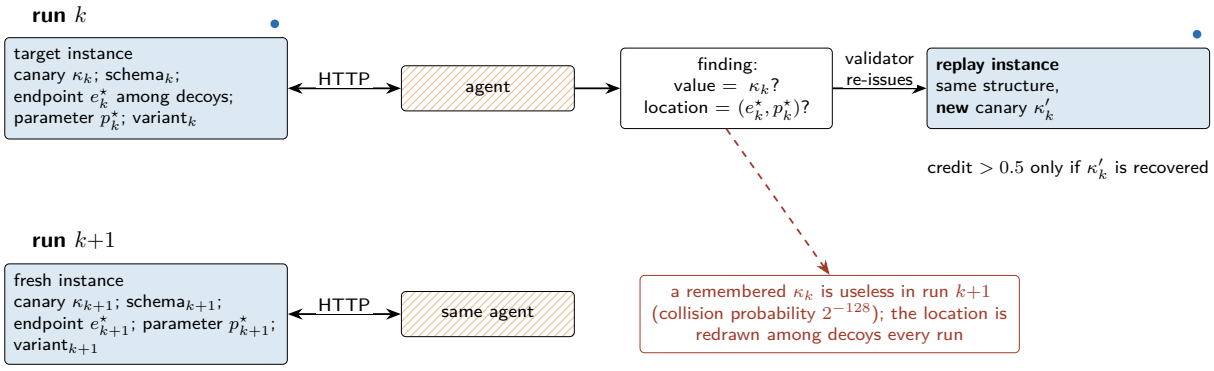
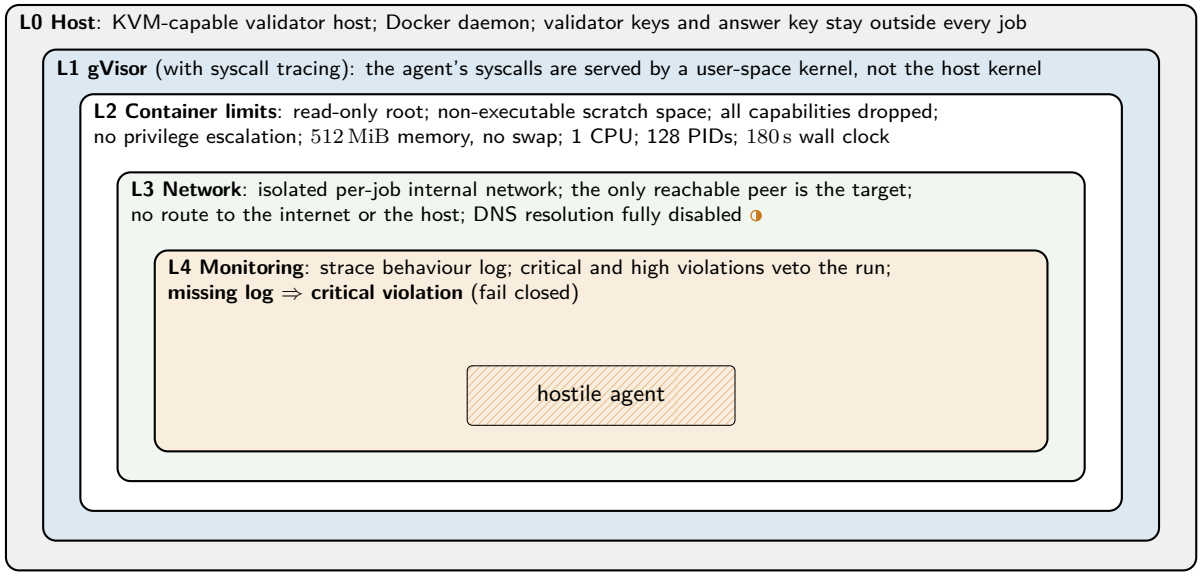



**Figure 5: Per-run randomization and replay** (concept, ●). Everything an agent could memorize is redrawn between runs, and credit above half requires reproducing the flaw on a fresh instance carrying a different secret. The figure is deliberately non-operational.




**Only outputs:** findings file (≤1 MiB, ≤200 items, regular file only); strace log (read by the validator); HTTP traffic to the target.

**Evidence:** a manual escape suite on one KVM host (reported by the developers, not independently verified).

**Figure 6: Containment layers** (● unless marked). The agent's only outputs are a size-capped findings file, its behaviour trace, and HTTP traffic to its own target. If the trace is missing, the run is treated as unsafe rather than passed.

## 5.4 Containment

Because the agent is adversarial code rewarded for attacking, it runs behind layered containment (Fig. 6). No single boundary is trusted alone.

- A gVisor sandbox interposes a user-space kernel between the agent and the host.
- Container limits strip capabilities and privileges, make the root file system read-only, and cap memory, CPU, processes, and wall-clock time.
- An isolated per-job network leaves the target as the only reachable peer, with no route to the internet or the host, and DNS disabled (◑ IN DEVELOPMENT).
- System-call monitoring records the agent's behaviour. Critical or high-severity actions, such as reaching for a container socket, host paths, or cloud-metadata addresses, veto the run. Monitoring *fails closed*: if a run produced no trace, it is treated as unsafe.

A vetoed run scores zero regardless of what the agent found: a high task score never excuses misbehaviour.

## 5.5 Scoring

Figure 7 shows how one run becomes an on-chain weight.

**Run score.** For a run on challenge $x$, define three events from the agent's findings $F$: found (some finding carries exactly $\kappa$), replay (its reproduction recovers the new canary on a fresh instance), and located (the matched finding names $e^\star$ and $p^\star$). Let $FP$ count the *other* findings placed on endpoints known to be safe. The bounded ladder and task score are RULE:

$$b(x) = \begin{cases} 0 & \neg\text{found} \\ 0.5 & \text{found} \wedge \neg\text{replay} \\ 0.8 & \text{found} \wedge \text{replay} \wedge \neg\text{located} \\ 1.0 & \text{found} \wedge \text{replay} \wedge \text{located} \end{cases} \qquad T(x) = \max\bigl(0,\ b(x) - \lambda\, FP\bigr), \quad \lambda = 0.10. \tag{2}$$

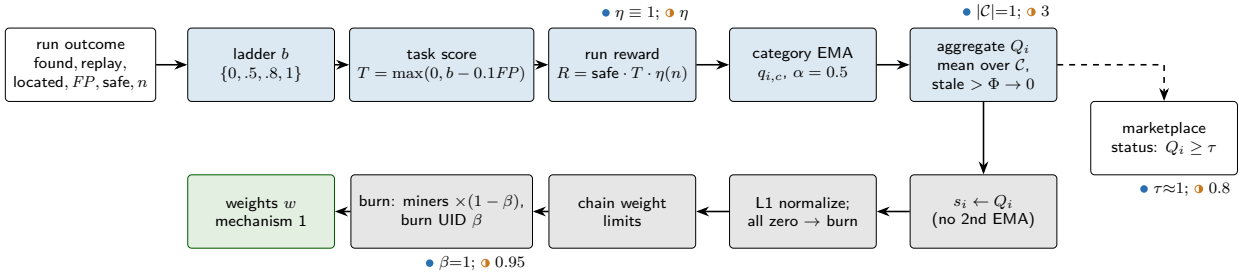



**Figure 7: From one run to a weight.** The task score is bounded by the canary ladder, vetoed by safety, and (◑ IN DEVELOPMENT) scaled by efficiency. Each run updates one category cell; the aggregate averages all active categories and counts a stale or untested one as zero. The marketplace gate reads the same aggregate that feeds the weights.

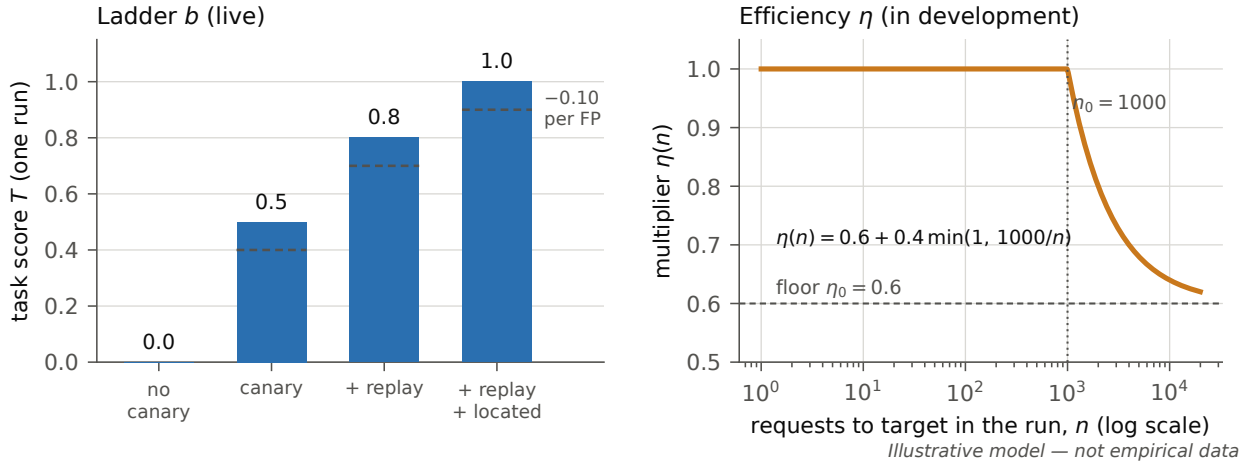



**Figure 8: Score ladder and efficiency factor** (formula plots; *illustrative model, not empirical*). Left: the four ladder levels of (2), with the drop after one false positive. Right: the efficiency multiplier of (3); agents that solve within 1000 requests keep full credit, and heavy volume is discounted toward 0.6. The efficiency factor is ◑.

The canary alone earns half credit; full credit requires a reproduced, precisely located flaw. Each false positive on a known-safe endpoint costs 0.10, so flooding findings across the surface is never free.

**Run reward.** With the safety verdict $\text{safe} \in \{0, 1\}$ and request count $n$, the reward is RULE:

$$R(x) = \begin{cases} (\text{not scored}) & \text{validator-side error} \\ 0 & \neg\text{safe} \ \text{ or no valid findings} \\ T(x) \cdot \eta(n) & \text{otherwise,} \end{cases} \qquad \eta(n) = \eta_0 + (1 - \eta_0)\,\min\Big(1, \tfrac{n_0}{n}\Big), \quad (3)$$

with $\eta_0 = 0.6$ and $n_0 = 1000$. The efficiency factor $\eta$ (◑ IN DEVELOPMENT) rewards a surgical agent over a brute-forcing one; today a run is scored by $T$ alone (Fig. 8). A validator-side error yields no judgement, so an outage never records a zero against an innocent miner.

**Breadth, freshness, and weight.** A single run tests one category, so the validator keeps a per-miner, per-category moving average, seeded on the first observation and updated with weight $\alpha = 0.5$ RULE:

$$q_{i,c} \leftarrow \alpha\, R + (1 - \alpha)\, q_{i,c}. \quad (4)$$

**Table 3: Mechanism-1 parameters.** "Live" is the value running today; "In dev." is the value on the branch being tested. Several values are configurable; the table gives the defaults.

| Symbol | Meaning | Live | In dev. |
|---|---|---|---|
| $\lambda$ | false-positive penalty | 0.10 | 0.10 |
| $b$ | ladder levels | $\{0, .5, .8, 1\}$ | same |
| $\eta_0, n_0$ | efficiency floor, target requests | — | 0.6, 1000 |
| $\alpha$ | category moving-average weight | 0.5 | 0.5 |
| $\Phi$ | freshness window | 1,800 s | 7,200 s |
| $\mathcal{C}$ | active categories | {sqli} | {sqli, cmdi, lfi} |
| $\beta$ | burn fraction (mainnet) | 1.0 | — |
| $\tau$ | marketplace gate | $\approx 1.0$ | 0.8 |
| — | round interval; budget | 300 s; 240 s | same |
| — | agent time limit | 180 s | 180 s |

The miner's aggregate averages over *all* active categories, counting any category that is untested or older than the freshness window $\Phi$ as zero:

$$Q_i = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} q_{i,c} \cdot \mathbb{1}[\text{fresh}] \,. \tag{5}$$

This rewards breadth: a one-trick agent caps at $1/|\mathcal{C}|$, while an all-rounder approaches 1. The aggregate becomes the miner's score, is normalized across miners, and is split with the burn account at fraction $\beta$:

$$w_i = (1 - \beta) \frac{Q_i}{\sum_j Q_j} \quad (i \neq \text{burn}), \qquad w_{\text{burn}} = \beta. \tag{6}$$

With $\beta = 1$ on mainnet today, the miner term vanishes and all emission burns during bootstrap. Table 3 collects the parameters.

### 5.6 Properties

We state the scoring guarantees as conditional propositions. Each holds under the assumptions of Section 3; none is a theorem about the live network, which has not yet run under reward.

**Proposition P1** (Fabrication resistance)**.** RULE If the canary is uniform on $\{0,1\}^{128}$ and reachable only through the planted flaw (Assumption A4), then an agent that never exercises the flaw triggers `found` with probability at most $m \cdot 2^{-128}$ per run, where $m \leq 200$ is the number of findings. A fabricated report earns essentially nothing.

**Proposition P2** (Memorization resistance)**.** RULE Because the canary is fresh each run and replay uses a new canary on a fresh instance, a cached or guessed secret cannot raise a score above 0.5. *Caveat:* for the blind variant the validator re-derives the secret itself, so there replay attests the target, not the agent's procedure.

**Proposition P3** (Breadth requirement)**.** RULE Since $Q_i \leq f_i/|\mathcal{C}|$ where $f_i$ is the number of fresh category cells, reaching the gate $Q_i \geq \tau$ forces at least $\lceil \tau|\mathcal{C}| \rceil$ fresh categories. With three categories and a gate of 0.8, a single-category specialist caps at 1/3 and cannot be listed.

**Proposition P4** (Copy resistance, limited)**.** RULE Encryption hides an agent from peers in transit, and digest de-duplication rejects a byte-identical resubmission on a validator that already recorded it. The limits are real: de-duplication is per-validator, a one-layer change yields a new digest, and the mechanism is not Sybil-proof.

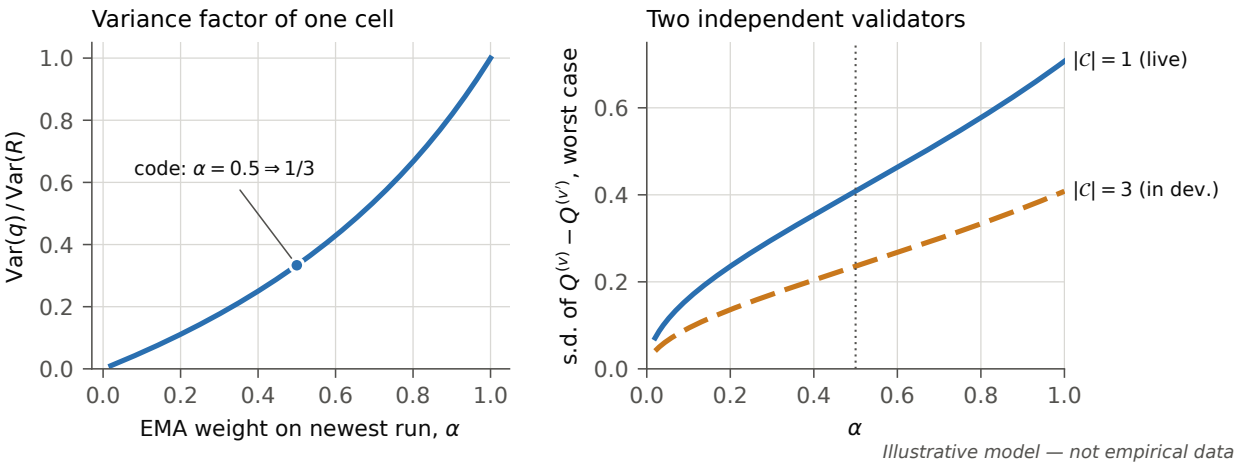



**Figure 9: Validator disagreement under the moving-average model** (*illustrative, not measured*; Proposition P5). Left: the stationary variance factor $\alpha/(2-\alpha)$; the live $\alpha = 0.5$ keeps one third of the per-run variance. Right: the worst-case spread between two honest validators, which breadth across categories reduces.

**Table 4: Threat summary for mechanism 1.** Status marks where the mitigation lives.

| Threat | Mitigation | Residual | St. |
|---|---|---|---|
| Fabricated finding | exact per-run canary match (Proposition P1) | depends on target integrity (Assumption A4) | ● |
| Memorized secret | fresh secret + replay on a new instance (Proposition P2) | blind variant attests location, not procedure | ● |
| Endpoint spraying | −0.10 per known-safe false positive | unknown endpoints unpenalized | ● |
| Copying an agent | encryption in transit; digest de-dup (Proposition P4) | per-validator; not Sybil-proof | ● |
| Hostile agent escape | layered gVisor containment, fail-closed monitor | third-party sandbox bugs; escape suite manual | ● |
| Exfiltration via DNS | isolated network; DNS disabled | stronger on the in-development branch | ◑ |
| Validator leak / laziness | commit–reveal weights; Yuma clipping | validator sees secrets and agents | ● |

**Proposition P5** (Validator agreement is a sampling question)**.** MODEL Independent validators draw independent challenges, so a category cell is a moving average of near-Bernoulli outcomes with stationary variance $\frac{\alpha}{2-\alpha}\sigma^2$, i.e. $\sigma^2/3$ at $\alpha = 0.5$. Two validators' aggregates then differ by $O(\sqrt{1/|\mathcal{C}|})$ (Fig. 9). Whether this stays within Yuma's tolerance is empirical (Section 7).

**Threats and residual risk.** Table 4 summarizes the main threats to mechanism 1. The recurring residual is validator trust: a validator sees plaintext agents and answer keys, and nothing but Yuma limits a validator that leaks or deviates.

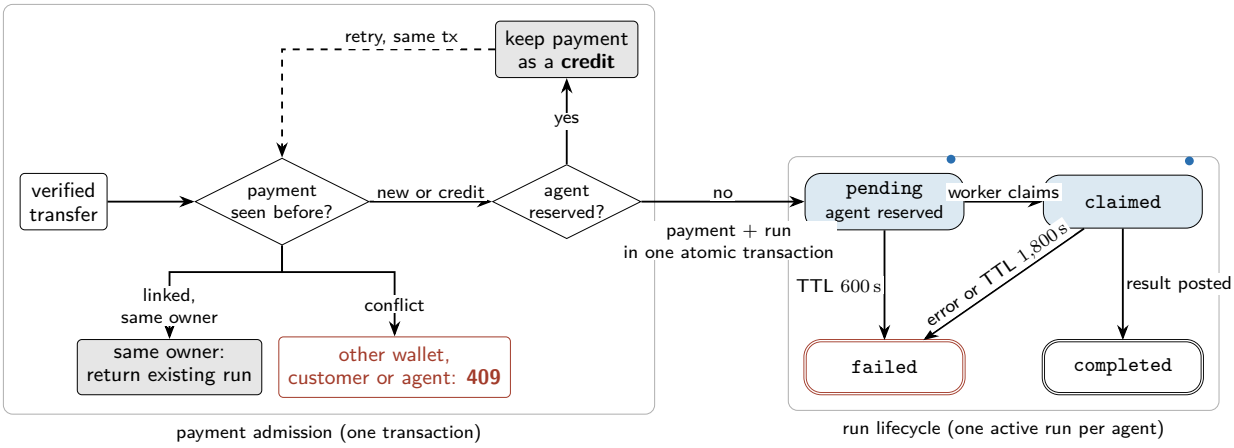



**Figure 10: Payment admission and run lifecycle** (●). A transfer is admitted at most once: a repeat of the same payment either returns the run it already funds or is refused; while an agent is busy, the payment is held as a credit and redeemed later. Time-to-live limits bound how long an abandoned reservation can hold an agent.

# 6 The Security Agent Marketplace

The marketplace turns a benchmark score into a service. Agents that clear the mechanism-1 gate are listed, and signed-in customers can commission audits of targets they own. It is the bridge from measured capability to paid work.

## 6.1 Audit lifecycle

A customer signs in, chooses a listed agent and a target (a live URL or a code repository), and attests that they are authorized to test it. The marketplace quotes a price in USD, the customer pays in TAO, and the marketplace verifies the payment on-chain before a run is created. Each agent serves at most one audit at a time, so a payment reserves the agent exclusively (Fig. 10).

## 6.2 Pricing and settlement

Audits are denominated in USD but settled in TAO (Fig. 11). The marketplace reads a TAO/USD rate from a price source (with a fallback), caches it briefly, and converts the fixed USD price into a RAO quote (1 TAO is $10^9$ RAO):

$$P_{\text{rao}} = \left\lceil \frac{P_{\text{USD}}}{\rho} \cdot 10^9 \right\rceil \text{ RAO}. \tag{7}$$

The buyer's wallet signs a transfer to the marketplace's payment address and waits for block inclusion. The marketplace then verifies on-chain that the transfer is on the expected network, that it succeeded, that its sender and recipient match, and that the amount is at least

$$A_{\min} = \left\lfloor \frac{P_{\text{rao}}\,(10^4 - \delta_{\text{bps}})}{10^4} \right\rfloor, \qquad \delta_{\text{bps}} = 300, \tag{8}$$

a tolerance band that absorbs small rate movement between quote and payment. Only then are the payment record and the audit run written together, in one atomic step, so a verified transfer maps to exactly one run and cannot be replayed for another.

## 6.3 Audit execution

The audit worker (◑ IN DEVELOPMENT) claims a paid run, checks that the chosen agent is still listed, and runs the audit in the same contained sandbox used for benchmarking (Fig. 12). Live-URL

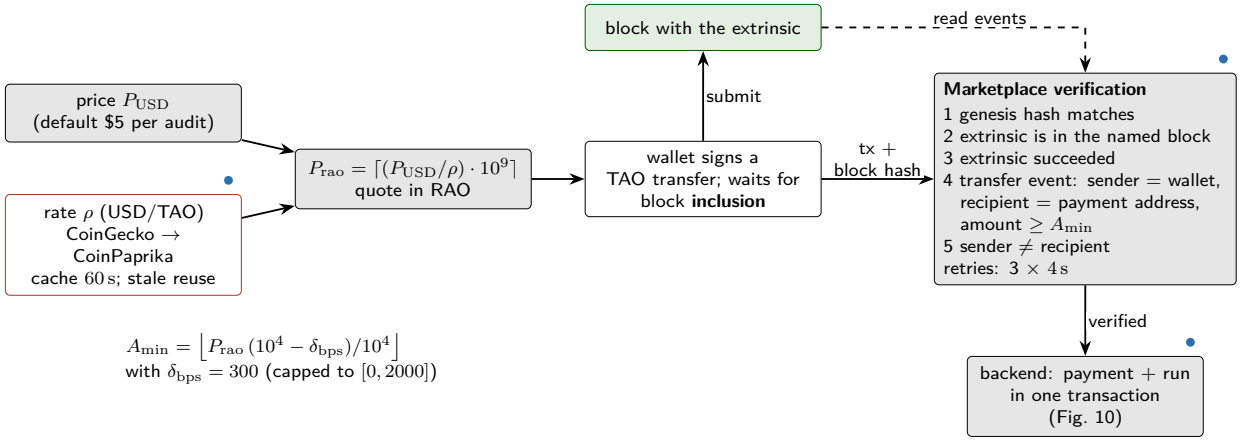



**Figure 11: Pricing and settlement** (●). A USD price becomes a TAO quote through an off-chain rate (red border: a trusted, fallible input). The customer pays on-chain, the marketplace checks the transfer against the expected network, parties, and minimum amount, and only then commits the run.

targets pass an SSRF gate, with every resolved address checked to be public and traffic forced through a proxy pinned to the vetted address; repository targets are built and run with no network egress, and static scanners run alongside. A result is posted back to the customer's run. The marketplace is thus an evaluation-gated service with on-chain settlement: the mechanism-1 score decides which agents may be commissioned, and payment is bound to delivery through the atomic admission of Fig. 10.

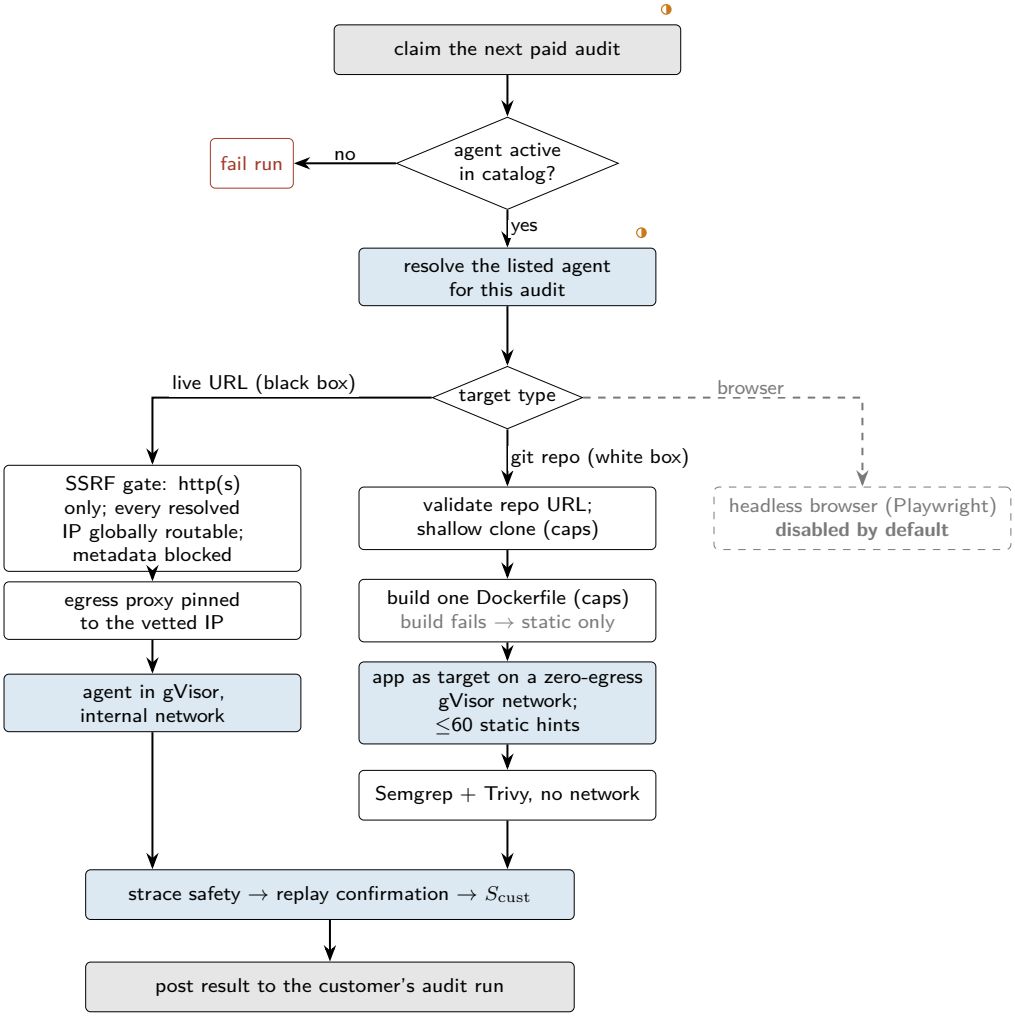



**Figure 12: Customer-audit flow** (◑). Every target type has a network gate: an SSRF check with a pinned proxy for live URLs, zero egress for repository builds, and a browser path that is off by default.

# 7 Current Status and Evaluation Plan

**What exists today.** Both mechanisms run on the subnet, and the marketplace serves a catalog of agents scored on testnet. Mainnet incentive is fully burned while the subnet bootstraps, so neither reward mapping has yet operated under real emission. We report *no* benchmark results: the figures in this paper are formula plots labelled as models, not measurements. What we can state is that the mechanisms are implemented as described, that the settlement logic behaves as specified under its own tests, and that early testnet evaluations have produced capability scores for security agents.

**Evaluation plan.** The claims that matter most are empirical and are the subject of ongoing evaluation. We frame them as research questions, each tied to a property above.

**RQ1** Does the score resist fabrication, memorization, spraying, and copying against red-team agents, and is replay sound for every variant (Propositions P1, P2 and P4)?

**RQ2** How much do independent validators disagree given per-run random challenges, and does the disagreement stay within Yuma's tolerance (Proposition P5)?

**RQ3** Does the sandbox contain hostile agents in a repeatable test suite, and at what overhead (Section 5.4)?

**RQ4** Do breadth, freshness, and burn behave as designed at realistic scale once emission is non-zero

(Section 5.5)?

**RQ5** Does a high benchmark score predict useful audits on held-out applications (Section 6)?

All evaluation targets are deliberately vulnerable applications run inside the operator's own sandbox under written rules of engagement [37, 38]; no third-party system is tested, and held-out applications are chosen so they are not used to tune agents [7, 39]. Until these results exist, the propositions remain conditional claims, not measured guarantees.

# 8 Limitations and Roadmap

**Limitations.** The honest boundaries of the current system are as follows, each tied to an assumption from Section 3.

- *Validator trust* (Assumption A2). A mechanism-1 validator sees every agent in plaintext and holds every answer key. Yuma limits how far one validator can move weights, but nothing stops off-chain leakage.
- *Target integrity* (Assumption A4). Fabrication resistance assumes the canary is reachable only through the planted flaw; each target design must be audited against this.
- *Settlement* (Assumption A6). Payment is accepted on block inclusion, not finality, and the price depends on an off-chain source; both are trusted inputs.
- *Narrow measurement.* The score covers synthetic, single-flaw targets in a small set of categories; it says nothing about missed vulnerabilities, multi-step logic flaws, or remediation quality.

**Roadmap (○ PLANNED).** The intended direction follows from these limits: lower the burn so the incentive mechanisms operate under real reward; broaden the vulnerability categories; strengthen replay confirmation across variants; and reduce the trust placed in a single validator and the operator. Each step requires the evidence set out in Section 7 before it can be claimed as delivered.

**Conclusion.** TAOStatus pays for two commodities on one subnet under the same principle: reward only what was demonstrably delivered. Mechanism 0 pays for LLM capacity by measured, epoch-lagged usage. Mechanism 1 pays for security capability by replayable proof-of-exploit, mapped to weights through a bounded, breadth-aware aggregate and used to gate a settled marketplace. The design is honest about its boundaries: the central proof-of-exploit property holds by construction for a single run, while its behaviour across validators, against adaptive adversaries, and as a predictor of real audit value is what the evaluation plan is built to establish.

# References


[1] A. Doupé, M. Cova, and G. Vigna. "Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners". In: *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2010)*. Vol. 6201. Lecture Notes in Computer Science. 2010, pp. 111–131. DOI: 10.1007/978-3-642-14215-4_7.

[2] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. "State of the Art: Automated Black-Box Web Application Vulnerability Testing". In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 332–345. DOI: 10.1109/SP.2010.27.

[3] R. Fang, R. Bindu, A. Gupta, Q. Zhan, and D. Kang. *LLM Agents can Autonomously Hack Websites*. 2024. arXiv: 2402.06664.

[4] R. Fang, R. Bindu, A. Gupta, and D. Kang. *LLM Agents can Autonomously Exploit One-day Vulnerabilities*. 2024. arXiv: 2404.08144.

[5] G. Deng, Y. Liu, V. Mayoral-Vilches, et al. "PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing". In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 847–864. arXiv: 2308.06782.

[6] C. Xu, S. Guan, D. Greene, and M.-T. Kechadi. *Benchmark Data Contamination of Large Language Models: A Survey*. 2024. arXiv: 2406.04244.

[7] S. Kapoor, B. Stroebl, Z. S. Siegel, N. Nadgir, and A. Narayanan. *AI Agents That Matter*. 2024. arXiv: 2407.01502.

[8] D. Manheim and S. Garrabrant. *Categorizing Variants of Goodhart's Law*. 2018. arXiv: 1803.04585.

[9] Learn Bittensor. *Yuma Consensus*. Learn Bittensor Developer Guides (web). Page footer: Last updated Aug 03, 2026. 2026.

[10] J. Steeves, A. Shaabana, Y. Hu, F. Luus, S. T. Liu, and J. D. Tasker-Steeves. *Incentivizing Intelligence: The Bittensor Approach*. Workshop paper; PDF footer: TSRML 2022 (Trustworthy and Socially Responsible Machine Learning) co-located with NeurIPS 2022. Landing page: https://bittensor.com/academia. 2022.

[11] Learn Bittensor. *Emission*. Learn Bittensor Developer Guides (web). Page footer: Last updated Aug 03, 2026. 2026.

[12] Y. Rao, Dr. Nick, and 0xcacti. *Bittensor: A Peer-to-Peer Intelligence Market (Dynamic TAO whitepaper)*. Web whitepaper. Undated on the page; page title "dTAO Whitepaper".

[13] Learn Bittensor. *Multiple Incentive Mechanisms Within Subnets*. Learn Bittensor Developer Guides (web). Page footer: Last updated Aug 03, 2026. 2026.

[14] Learn Bittensor. *Commit Reveal*. Learn Bittensor Developer Guides (web). Page footer: Last updated Aug 03, 2026. 2026.

[15] Learn Bittensor. *The Weight Copying Problem*. Learn Bittensor Developer Guides (web). Page footer: Last updated Aug 03, 2026. 2026.

[16] E. Lui and J. Sun. *Bittensor Protocol: The Bitcoin in Decentralized Artificial Intelligence? A Critical and Empirical Analysis*. arXiv comment: MARBLE 2025. 2025. arXiv: 2507.02951.

[17] Y. Rao, J. Steeves, A. Shaabana, D. Attevelt, and M. McAteer. *BitTensor: A Peer-to-Peer Intelligence Market*. arXiv v1 2020-03-09; v2 2021-03-10; v3 2021-11-10 is a withdrawal notice (author-marked obsolete and incomplete; author D. Attevelt removed). Cite v1 or v2 and disclose status. 2020. arXiv: 2003.03917.

[18] A. K. Zhang, N. Perry, R. Dulepet, et al. "Cybench: A Framework for Evaluating Cybersecurity Capabilities and Risks of Language Models". In: *International Conference on Learning Representations (ICLR)*. ICLR 2025 Oral (per arXiv comment). 2025. arXiv: 2408.08926.

[19] M. Shao, S. Jancheska, M. Udeshi, et al. "NYU CTF Bench: A Scalable Open-Source Benchmark Dataset for Evaluating LLMs in Offensive Security". In: *Advances in Neural Information Processing Systems 37 (Datasets and Benchmarks Track)*. 2024, pp. 57472–57498. DOI: 10.52202/079017-1832. arXiv: 2406.05590.

[20] Y. Zhu, A. Kellermann, D. Bowman, et al. "CVE-Bench: A Benchmark for AI Agents' Ability to Exploit Real-World Web Application Vulnerabilities". In: *Proceedings of the 42nd International Conference on Machine Learning*. Vol. 267. Proceedings of Machine Learning Research. 2025, pp. 79850–79867. arXiv: 2503.17332.

[21] A. K. Zhang, J. Ji, C. Menders, et al. *BountyBench: Dollar Impact of AI Agent Attackers and Defenders on Real-World Cybersecurity Systems*. 2025. arXiv: 2505.15216.

[22] J. Yang, A. Prabhakar, K. Narasimhan, and S. Yao. "InterCode: Standardizing and Benchmarking Interactive Coding with Execution Feedback". In: *Advances in Neural Information Processing Systems 36 (Datasets and Benchmarks Track)*. 2023, pp. 23826–23854. DOI: 10.52202/075280-1035. arXiv: 2306.14898.

[23] C. E. Jimenez, J. Yang, A. Wettig, et al. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" In: *International Conference on Learning Representations (ICLR)*. 2024. arXiv: 2310.06770.

[24] OWASP Foundation. *OWASP Benchmark Project*. Web page.

[25] T. Boland and P. E. Black. "Juliet 1.1 C/C++ and Java Test Suite". In: *Computer* 45.10 (2012), pp. 88–90. DOI: 10.1109/MC.2012.345.

[26] NIST. *Software Assurance Reference Dataset (SARD)*. Web page.

[27] N. Miller, P. Resnick, and R. Zeckhauser. "Eliciting Informative Feedback: The Peer-Prediction Method". In: *Management Science* 51.9 (2005), pp. 1359–1373. DOI: 10.1287/mnsc.1050.0379.

[28] D. Prelec. "A Bayesian Truth Serum for Subjective Data". In: *Science* 306.5695 (2004), pp. 462–466. DOI: 10.1126/science.1102081.

[29] J. Teutsch and C. Reitwießner. *A scalable verification solution for blockchains*. Reprinted: Aspects of Computation and Automata Theory with Applications, NUS IMS Lecture Notes Series, World Scientific, 2023, pp. 377–424, doi:10.1142/9789811278631_0015. 2019. arXiv: 1908.04756.

[30] A. Arun, A. St. Arnaud, A. Titov, et al. *Verde: Verification via Refereed Delegation for Machine Learning Programs*. 2025. arXiv: 2502.19405.

[31] B. Parno, J. Howell, C. Gentry, and M. Raykova. "Pinocchio: Nearly Practical Verifiable Computation". In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 238–252. DOI: 10.1109/SP.2013.47.

[32] T. Maillart, M. Zhao, J. Grossklags, and J. Chuang. "Given enough eyeballs, all bugs are shallow? Revisiting Eric Raymond with bug bounty programs". In: *Journal of Cybersecurity* 3.2 (2017), pp. 81–90. DOI: 10.1093/cybsec/tyx008.

[33] M. Zhao, J. Grossklags, and P. Liu. "An Empirical Study of Web Vulnerability Discovery Ecosystems". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015, pp. 1105–1117. DOI: 10.1145/2810103.2813704.

[34] gVisor Authors. *What is gVisor?* Web page.

[35] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "The True Cost of Containing: A gVisor Case Study". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA, July 2019.

[36] Anjali, T. Caraza-Harter, and M. M. Swift. "Blending Containers and Virtual Machines: A Study of Firecracker and gVisor". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. 2020, pp. 101–113. DOI: 10.1145/3381052.3381315.

[37] K. Scarfone, M. Souppaya, A. Cody, and A. Orebaugh. *Technical Guide to Information Security Testing and Assessment*. Tech. rep. SP 800-115. National Institute of Standards and Technology, Sept. 2008.

[38] OWASP Foundation. *Web Security Testing Guide*. Web page.

[39] C. Dwork, V. Feldman, M. Hardt, T. Pitassi, O. Reingold, and A. Roth. "The Reusable Holdout: Preserving Validity in Adaptive Data Analysis". In: *Science* 349.6248 (2015), pp. 636–638. DOI: 10.1126/science.aaa9375.