Consensix Labs

AI-Verified Software Delivery Escrow

A Proof of Concept for Using AI Agents as Third-Party Verifiers in Smart Contracts

Table of Contents

Executive Summary

When one company hires another to build software, a fundamental question arises: *how do you know the work is actually done?* Today, this is resolved through manual review, meetings, and trust — processes that are slow, expensive, and subjective. Disputes over whether a deliverable meets its specification are common and costly.

This paper presents a proof of concept that combines two technologies to address this problem: **blockchain-based smart contracts** for trustless financial escrow, and **AI agents** for automated verification of software deliverables. The client deposits funds into a smart contract. The contractor submits their work. An AI agent evaluates the deliverable against the agreed requirements — running the software, testing it, and scoring its compliance. If the work meets the threshold, funds are released automatically. If it clearly fails, the contractor can fix and resubmit. If the AI is uncertain, the case is escalated to a human arbiter with a detailed report.

The result is a system that is faster and cheaper than manual review for clear-cut cases, while preserving human judgment for ambiguous ones. This paper describes the concept, discusses its advantages and limitations, presents a working implementation, and shares results from testing it with intentionally varied deliverables.

1. Introduction

The Problem

Software delivery disputes are a persistent friction in the technology industry. A client writes a specification. A contractor builds software against it. At the point of delivery, disagreements arise: *Does the API handle edge cases? Is the feature complete? Does it match the design?* These disputes consume time, strain relationships, and often require expensive third-party arbitration.

The root cause is verification. Unlike physical goods — where you can inspect a shipment and confirm it matches the order — software is complex, layered, and its quality is partly subjective. Verification today relies on human reviewers who may be slow, biased, or unfamiliar with the full specification.

The Opportunity

Two technologies have matured to the point where a better approach is feasible:

Smart contracts are programs that run on a blockchain and execute automatically when conditions are met. They are commonly used for financial escrow: funds are locked in the contract and released only when predetermined conditions are satisfied. No trusted intermediary is needed — the blockchain itself enforces the rules. Smart contracts are already used to escrow billions of dollars in decentralized finance.

AI agents — specifically large language models (LLMs) like GPT-4 — have demonstrated remarkable ability to understand software specifications, analyze code, and evaluate whether requirements are met. They can read an API specification, examine test results, and produce a structured assessment that would take a human reviewer hours.

The idea explored in this paper is straightforward: *What if we used an AI agent as the verifier in a smart contract escrow arrangement?* The AI becomes a neutral third party — faster and cheaper than a human, available 24/7, and consistent in its evaluations.

2. Background

Smart Contracts and Escrow

A smart contract is a piece of code deployed on a blockchain that executes automatically according to its programmed rules. Once deployed, nobody — not even the person who created it — can alter its behavior. This immutability is what makes smart contracts trustworthy: both parties can inspect the code and know exactly what will happen under any circumstance.

In an escrow arrangement, the smart contract holds funds deposited by the client. The contract defines the conditions under which those funds are released to the contractor (or refunded to the client). The key advantage over traditional escrow is that no bank, lawyer, or escrow service needs to be trusted — the code is the arbiter.

The challenge is that smart contracts can only act on information that exists on the blockchain. They cannot browse the web, run tests, or evaluate software. This is where **oracles** come in.

Oracles: Connecting Blockchains to the Real World

An oracle is a service that feeds external information to a smart contract. For example, a price oracle tells a DeFi contract the current exchange rate. A weather oracle tells an insurance contract whether a hurricane occurred. The oracle bridges the gap between the blockchain's isolated world and external reality.

In our system, the AI verifier acts as an oracle: it evaluates a real-world event (the delivery of software) and reports the result to the smart contract, which then acts on it (releasing or withholding funds).

AI Agents for Software Verification

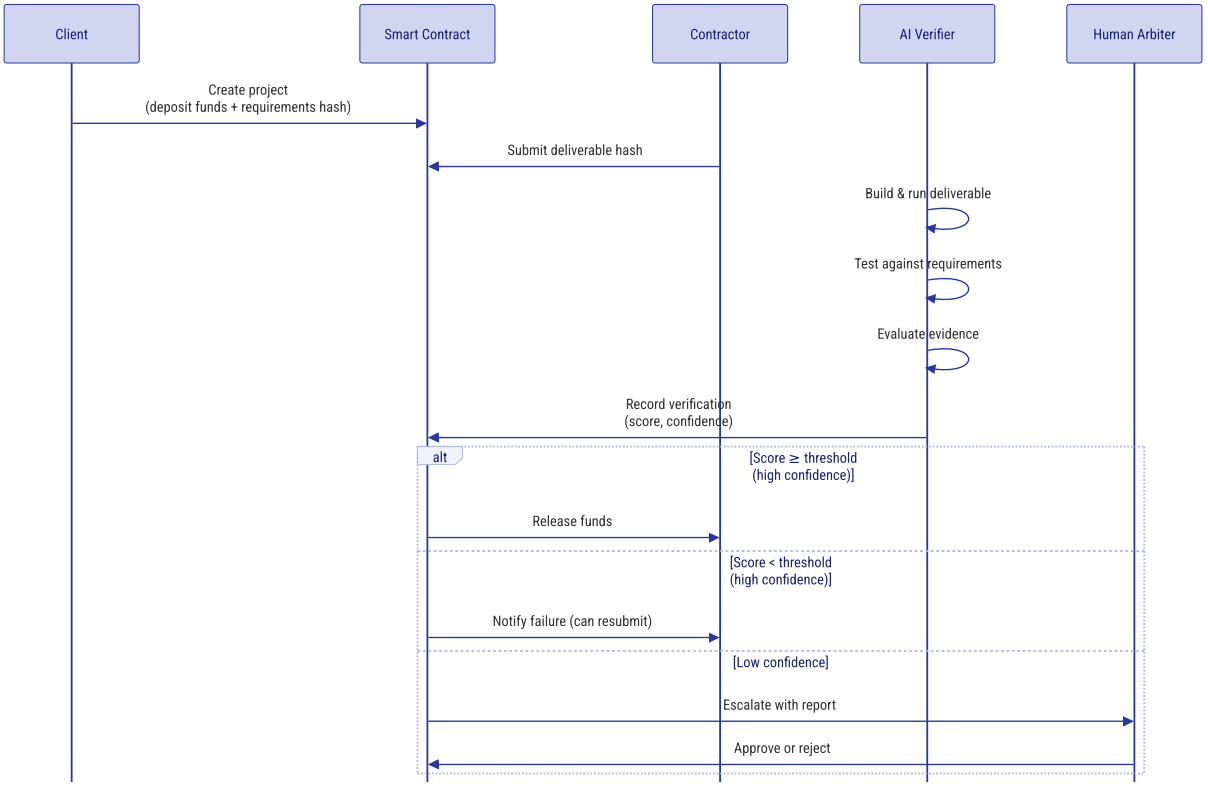
Modern LLMs can perform tasks that were previously possible only for human experts:

- **Read and understand** API specifications, requirements documents, and test suites.
- **Analyze** test results and determine whether failures are critical or cosmetic.
- **Evaluate** whether a deliverable matches the intent of a specification, not just its letter.
- **Produce** structured, auditable reports explaining their reasoning.

This makes them suitable candidates for the oracle role in software delivery verification, particularly for cases where the requirements can be tested programmatically (API conformance, test suite passage, etc.).

3. The Concept

How It Works



The flow has five stages:

1. **Agreement.** The client and contractor agree on requirements, a price, a pass threshold (e.g., "the deliverable must score at least 70/100"), and a deadline. The client deposits funds into the smart contract.

2. **Submission.** The contractor builds the software and submits it. A cryptographic hash of the deliverable is recorded on the blockchain, creating a tamper-proof record of exactly what was submitted.

3. **Verification.** The AI verifier retrieves the deliverable, builds and runs it, tests it against the requirements, and asks an LLM to evaluate the results. The evaluation produces a score (0–100), a confidence level (high or low), and a detailed report.

4. **Resolution.** The smart contract acts on the AI's result:

    - **High confidence, passing score** → Funds are automatically released to the contractor.
    - **High confidence, failing score** → The contractor is notified and can fix and resubmit.
    - **Low confidence** → The case is escalated to a human arbiter, who receives the full evaluation report.

5. **Fallback.** If the project remains unresolved past its deadline, the client can reclaim their funds.

Note that the smart contract's decision is based solely on the **numeric score** and **confidence level** — not the AI's textual recommendation. The AI may recommend "fail" in its report, but if the score meets the threshold, the contract releases funds regardless. This is a deliberate design choice: the contract enforces a simple, deterministic rule (score vs. threshold), while the AI's richer assessment is available in the off-chain report for human review. As discussed in Section 7, this distinction has practical implications for threshold selection.

Advantages

Speed. Automated verification can complete in minutes rather than the days or weeks a manual review might take.

Cost. An AI evaluation costs a fraction of a human expert's time, making it practical even for small engagements.

Consistency. The same requirements are evaluated the same way every time. There is no reviewer fatigue or subjective drift.

Availability. The verifier runs 24/7 without scheduling constraints.

Audit trail. Every step — submission, verification, scoring — is recorded on the blockchain with cryptographic integrity.

Graduated autonomy. The system doesn't pretend the AI is infallible. Clear-cut cases are resolved automatically; ambiguous ones are escalated with a useful report that saves the human arbiter time.

Challenges and Limitations

Non-determinism. LLMs can produce different outputs for the same input. Two evaluations of the same deliverable might yield slightly different scores. This is mitigated by using low-temperature settings and structured prompts, but cannot be eliminated entirely.

Adversarial gaming. If real money is at stake, a contractor has incentive to game the evaluator — writing code that passes automated tests but is subtly broken in ways the AI doesn't detect. This is analogous to "teaching to the test" in education.

Spirit vs. letter. The most valuable human judgment in software delivery is assessing whether the deliverable matches the *intent* of the requirements, not just their technical specification. AI is better at the letter than the spirit, though this gap is narrowing.

Liability. If the AI makes a wrong call and funds are released for incomplete work (or withheld for complete work), the question of legal responsibility is currently unresolved. The escrow contract has no mechanism for appeal beyond the arbiter.

Verification integrity. In the current proof of concept, if the deliverable hash submitted on-chain doesn't match the artifact the verifier evaluates, this mismatch is detected but only logged off-chain. A production system should record verification attempts (including failures) on-chain to prevent accountability gaps. See the Technical Implementation section for details.

The Multi-Oracle Variant

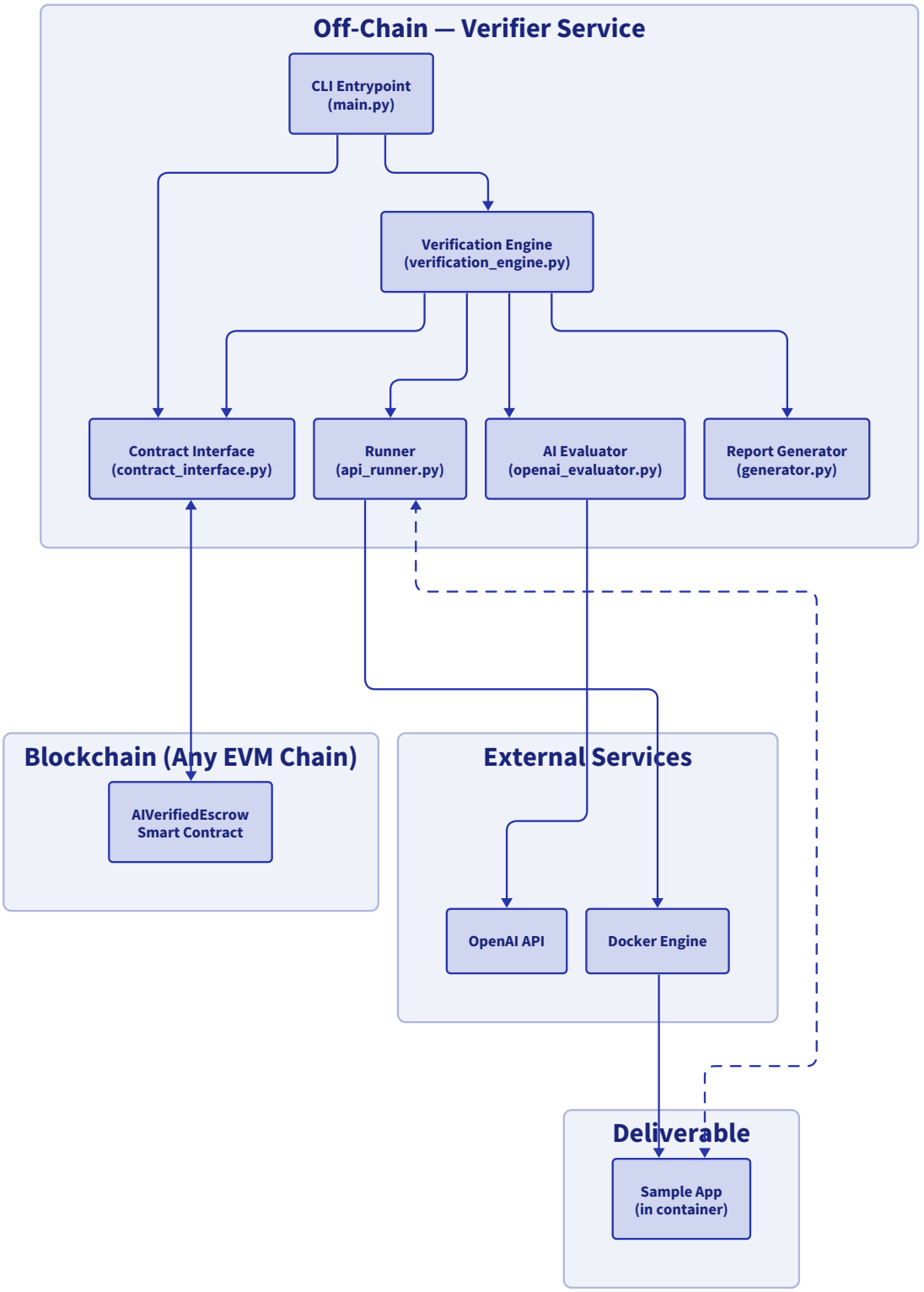
A natural extension of this system is to use multiple independent AI verifiers, each evaluating the same deliverable and submitting results to the contract. Funds would only be released if a majority (or a threshold) of verifiers agree. This provides:

- **Redundancy** against any single AI model's blind spots.
- **Resistance** to adversarial gaming (it's harder to fool three different models than one).
- **Consensus** that maps naturally to blockchain's existing consensus mechanisms.

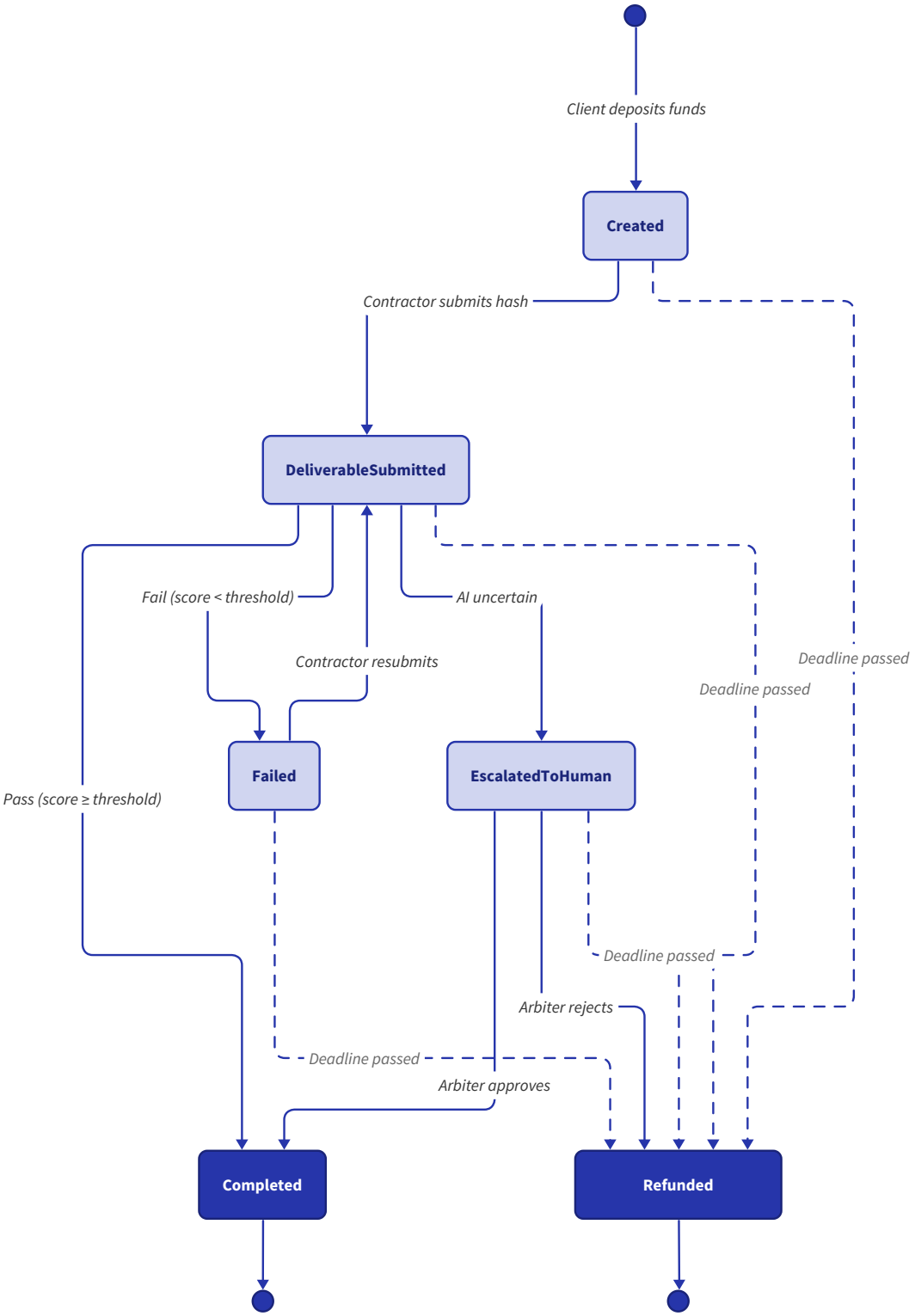
The trade-off is cost (multiple evaluations) and complexity (the contract must aggregate results). This variant is not implemented in the current PoC but the contract architecture supports it by allowing different verifier addresses per project.

4. Architecture

System Overview



Smart Contract State Machine



Data Flow

All bulk data (requirements documents, deliverable source code, full evaluation reports) lives **off-chain**. The blockchain stores only:

- **Content hashes** — SHA-256 fingerprints of the requirements and deliverable, so anyone can verify the off-chain data hasn't been tampered with.
- **Verification results** — score, confidence, AI model identifier, and a hash of the full report.
- **Financial state** — escrowed amount, release/refund events.

This design keeps gas costs low while preserving the blockchain's core value: an immutable, tamper-proof audit trail.

5. Technical Implementation

This section assumes familiarity with Solidity, Python, Docker, and REST APIs.

Smart Contract: `AIVerifiedEscrow.sol`

The contract is written in Solidity 0.8.28 and manages the full escrow lifecycle. Key design decisions:

Roles are per-project. Each project specifies its own client, contractor, verifier, and arbiter addresses. This means different projects can use different AI oracle providers, and the contract itself has no global admin or owner.

Verification history is preserved. The contract stores an array of `VerificationResult` structs per project, so if a contractor resubmits, the full history of verification attempts is available on-chain.

Confidence drives the outcome path. Rather than having a single pass/fail based on score, the contract uses a two-dimensional result (score × confidence) to determine the next state. This is what enables the graduated autonomy model.

Score determines fund release, not the AI's recommendation. The contract compares the numeric score against the threshold to decide whether to release funds. The AI also produces a textual recommendation ("release", "fail", or "escalate") in its off-chain report, but this is informational — the contract does not read it. This keeps the on-chain logic simple and deterministic while preserving richer context in the report for human review.

Reentrancy protection. Fund transfers zero the stored amount before executing the transfer, following the checks-effects-interactions pattern.

Extension point: Hash mismatch recording

The current contract assumes the verifier always receives a deliverable matching the on-chain hash. In a production system, the `recordVerification()` function should accept an additional parameter indicating whether the hash check passed, and the contract should have a `HashMismatch` state or event. This would be added to the `recordVerification` function:

```
// PRODUCTION: Add hash verification status
function recordVerification(
    uint256 projectId,
    uint8 score,
    Confidence confidence,
    string calldata summaryHash,
    string calldata modelId,
    bool hashVerified  // NEW: did the deliverable hash match?
) external onlyVerifier(projectId) {
    if (!hashVerified) {
        emit HashMismatchDetected(projectId, summaryHash);
        // Record the attempt but don't change project state
        // The contractor must resubmit with a matching deliverable
        return;
    }
    // ... existing verification logic
}
```

Verifier: Python Service

The verifier is a Python CLI built with Click that orchestrates the full pipeline. Its modular design allows new requirement types to be added by implementing a runner and an evaluator.

Contract Interface (`contract_interface.py`) — Compiles the Solidity contract using `py-solc-x`, deploys it via Web3.py, and provides typed wrappers for every contract function. Connection details come from environment variables, so the same code works against a local Hardhat node, a testnet, or mainnet.

Runners — Each runner knows how to start a deliverable, exercise it, and collect evidence. The `ApiRunner` builds a Docker image from the deliverable source, starts it in a container, parses the OpenAPI spec to generate test cases, hits each endpoint, and collects structured results.

Evaluators — Each evaluator takes evidence and requirements and produces a structured assessment using an AI model. The `OpenApiEvaluator` sends the OpenAPI spec and test evidence to OpenAI with a carefully designed prompt that requests a specific JSON output format with score, confidence, criteria breakdown, and recommendations.

Report Generator — Produces a JSON file (machine-readable, complete), a Markdown file (human-readable), and a log file (full console output). For escalated cases, the Markdown report includes all the information the human arbiter needs: which tests passed, which failed, what the AI was uncertain about, and instructions for calling the `arbitrate()` function.

Sample Project: Calculator API

The calculator API is a deliberately simple deliverable that makes verification outcomes easy to understand and validate. It implements four arithmetic endpoints (`/add`, `/subtract`, `/multiply`, `/divide`) specified by an OpenAPI 3.0 document.

Four variants test different failure modes:

Variant	Description
`complete`	All endpoints correct, edge cases handled
`missing_endpoint`	No `/multiply` route (returns 404)
`buggy`	`/subtract` returns `a+b` instead of `a-b`
`crash_on_edge_case`	Server crashes on division by zero

See Section 7 for the actual verification results.

6. Running the Demo

Prerequisites

- Docker and Docker Compose
- An OpenAI API key (GPT-4o recommended; GPT-4o-mini works for cheaper testing)
- One of:
  - The Ethereum Local Testing Starter Pack (recommended for local testing)
  - Access to a public EVM testnet (e.g. Sepolia) with funded accounts

Option A: Using the Starter Pack (Local)

Step 1: Start the blockchain.

```
cd /path/to/ethereum-local-testing-starter-pack
./eth.sh start
```

This starts two Hardhat nodes. The first node is available at `localhost:41545`.

Step 2: Create a verifier account.

The starter pack comes with two pre-funded accounts (client and contractor). The escrow system needs a third account for the AI verifier:

```
./eth.sh create-account 100
```

Note the address and private key from the output.

Step 3: Configure the environment.

```
cd /path/to/ai-verified-escrow
cp .env.example .env
```

Edit `.env`:

- Set `RPC_URL=http://host.docker.internal:41545`
- `PRIVATE_KEY_CLIENT` and `PRIVATE_KEY_CONTRACTOR` are pre-filled with the starter pack accounts
- Set `PRIVATE_KEY_VERIFIER` to the private key from step 2
- Set your `OPENAI_API_KEY`

Step 4: Deploy the smart contract.

```
docker compose run --rm verifier deploy
```

Copy the contract address from the output and add it to your `.env` file:

```
CONTRACT_ADDRESS=0x...
```

Step 5: Create a project.

```
docker compose run --rm verifier create-project \
  --sample-dir /app/samples/calculator_api \
  --amount 1.0
```

Note the project ID (should be `0` for the first project).

Step 6: Submit a deliverable.

```
# Try the complete (correct) variant first
docker compose run --rm verifier submit \
  --project-id 0 \
  --sample-dir /app/samples/calculator_api \
  --variant complete
```

Step 7: Run verification.

```
docker compose run --rm verifier verify \
  --project-id 0 \
  --sample-dir /app/samples/calculator_api \
  --variant complete
```

The verifier will build the deliverable, run tests, send evidence to OpenAI, record the result on-chain, and generate reports in the `reports/` directory.

Step 8: Check the result.

```
docker compose run --rm verifier status --project-id 0
```

Option B: Using a Public Testnet

1. Choose a testnet (Sepolia, Holesky, Arbitrum Sepolia, etc.).
2. Create three accounts (client, contractor, verifier) and fund them with testnet ETH from a faucet.
3. Get an RPC URL from a provider like Infura, Alchemy, or QuickNode.
4. Set the private keys and RPC URL in `.env`.
5. Follow Steps 4–8 above, replacing the `RPC_URL` accordingly.

Testing All Variants

A helper script runs all four variants end-to-end, creating a fresh project for each:

```
./run_all_variants.sh
```

Alternatively, you can test each variant manually. Each variant needs its own project since each project holds separate escrow funds:

```
# Example: test the buggy variant
docker compose run --rm verifier create-project \
  --sample-dir /app/samples/calculator_api --amount 1.0
docker compose run --rm verifier submit \
  --project-id 1 --sample-dir /app/samples/calculator_api --variant buggy
docker compose run --rm verifier verify \
  --project-id 1 --sample-dir /app/samples/calculator_api --variant buggy
```

7. Results

We ran all four Calculator API variants through the verification pipeline. Each variant was given its own escrow project with 1 ETH deposited and a pass threshold of 70/100. The AI evaluator (GPT-4o) assessed each deliverable against the OpenAPI specification, scoring five criteria: Endpoint Completeness, Correctness, Error Handling, Schema Compliance, and Stability.

Summary

Variant	Score	Confidence	Tests Passed	AI Recommendation	Contract Outcome
`complete`	100/100	High	16/16	Release	Completed (funds released)
`missing_endpoint`	75/100	High	13/16	Fail	Completed (funds released)*
`buggy`	65/100	High	13/16	Fail	Failed (can resubmit)
`crash_on_edge_case`	85/100	High	15/16	Fail	Completed (funds released)*

**The AI recommended "fail" but the score exceeded the 70/100 threshold, so the contract released funds. See "Observations" below.*

Variant: `complete`

This variant implements all four endpoints correctly with proper input validation and edge case handling.

The verifier ran 16 test cases and all passed. The AI gave a perfect score of 100/100 with high confidence, summarizing: *"The Calculator API meets all specified requirements. All endpoints are present and functioning correctly, with proper handling of edge cases and input validation."* Funds were released automatically.

Variant: `missing_endpoint`

This variant is missing the `/multiply` endpoint entirely — requests to it return 404.

The verifier detected 3 failures, all on the `/multiply` endpoint. The AI scored it 75/100, correctly identifying the gap: Endpoint Completeness scored 60/100 while all other criteria scored 90–100. The AI recommended failure, noting *"The multiplication endpoint is missing, resulting in failed tests."*

However, because the score of 75 exceeded the pass threshold of 70, the smart contract released funds to the contractor despite the AI's recommendation to fail.

Variant: `buggy`

This variant has a logic error in `/subtract` — it returns `a + b` instead of `a - b`. The bug is invisible for symmetric inputs like `0 - 0 = 0` but produces clearly wrong results for all other cases (e.g., `10 - 3` returns `13`).

The verifier caught 3 of 4 subtraction tests as failures (the `0 - 0` case passed because addition and subtraction produce the same result). The AI scored it 65/100, giving Correctness only 50/100 while all other criteria scored 100. It noted: *"The API has significant issues with the subtraction endpoint, which consistently returns incorrect results."*

With a score of 65 — below the 70 threshold — the contract correctly moved to the Failed state, allowing the contractor to fix the bug and resubmit.

Variant: `crash_on_edge_case`

This variant handles all normal inputs correctly but crashes with an unhandled `TypeError` when division by zero is attempted, returning a 500 error with a stack trace instead of the expected 400.

Only 1 of 16 tests failed. The AI scored it 85/100 — notably generous given that a server crash is typically a serious issue. Error Handling scored 60/100 while everything else scored 100. The AI identified the specific problem: *"The API fails to handle division by zero correctly, returning a 500 error instead of the expected 400."*

As with the missing endpoint variant, the score exceeded the 70 threshold and funds were released despite the AI's recommendation to fail.

Observations

Threshold selection is critical. The most striking result is that two clearly defective deliverables — one missing an entire endpoint, one that crashes on a common edge case — scored above the 70/100 threshold. Had this been a real escrow arrangement with these settings, funds would have been released for incomplete and unstable software. This underscores that the pass threshold must be set carefully, and that different projects may need very different thresholds. A threshold of 90 would have correctly rejected all three defective variants.

The AI's recommendation and score can diverge. The contract decides fund release based on the numeric score alone — it does not read the AI's textual recommendation. In two cases, the AI recommended "fail" but assigned a score above the threshold, resulting in fund release. This highlights a tension in the design: the AI's nuanced assessment (which considers the *nature* of failures, not just their count) may reach a different conclusion than the simple score-vs-threshold rule. Future iterations could address this by having the contract also consider the recommendation, or by prompting the AI to align its score more strictly with its recommendation.

The AI's scoring reflects proportionality, not binary pass/fail. The AI doesn't simply count test failures. It weighs the *severity* of each issue. A logic error in a core operation (65/100) is scored more harshly than a missing endpoint (75/100), which is scored more harshly than a single edge-case crash (85/100). This contextual judgment is the key value of using an AI evaluator rather than a simple pass-rate formula.

All results were high confidence. None of the four variants triggered the escalation-to-human path. This is expected for a well-specified API with clear test evidence — the OpenAPI spec makes requirements unambiguous, and the test results leave little room for interpretation. We would expect low-confidence results when dealing with fuzzier requirement types (visual design, natural language specifications) or ambiguous test evidence.

The buggy variant reveals a subtle edge case in testing. The subtraction test for `0 - 0` passed despite the bug because the buggy implementation (`a + b`) produces the same result as the correct one (`a - b`) when both operands are zero. This is a reminder that test case selection matters — adversarial or property-based testing would catch this more reliably.

Full verification logs, AI evaluation details, and per-test evidence for all four variants are available in Appendix B.

8. Future Directions

Additional Requirement Types

The current PoC demonstrates API verification against an OpenAPI spec. The architecture supports additional requirement types through new runner/evaluator pairs:

- **Test suite runner** — Execute a provided test suite (e.g., Jest, pytest) and evaluate results.
- **Visual design runner** — Capture screenshots and compare against design mockups using multimodal AI.
- **PDF specification runner** — Parse a PDF requirements document and generate test cases from natural language descriptions.
- **Performance runner** — Load-test the deliverable and evaluate response times against SLA requirements.

Multi-Oracle Consensus

As discussed in Section 3, running multiple independent verifiers and requiring consensus would improve reliability and adversarial resistance. Implementation would involve:

1. Allowing multiple verifier addresses per project.
2. A voting mechanism in the contract (e.g., 2-of-3 must agree).
3. Different AI models for each verifier to reduce correlated failures.
4. A reconciliation process when verifiers disagree.

Adversarial Robustness

Further research is needed on how contractors might game AI verifiers and how to defend against it. Potential approaches include:

- **Randomized test generation** — Don't only test the examples in the spec; generate novel edge cases.
- **Code analysis** — Have the AI review the source code, not just runtime behavior, to detect suspicious patterns.
- **Historical comparison** — Compare the contractor's deliverable against known-good implementations to detect superficial compliance.

On-Chain Verification Integrity

As noted throughout this paper, the PoC assumes the happy path where the deliverable always matches its on-chain hash. A production system should:

1. Record all verification attempts on-chain, including hash mismatches.
2. Track timing — when was the deliverable submitted, when did verification begin, when did it complete.
3. Support challenge mechanisms where either party can dispute the verifier's integrity.

Legal and Regulatory Considerations

Before deploying this system with real funds, several legal questions must be addressed:

- **Liability** — Who is responsible if the AI makes an incorrect verification? The oracle operator? The contract deployer? The AI provider?
- **Enforceability** — Are smart contract escrow arrangements legally enforceable in relevant jurisdictions?
- **Regulatory compliance** — Does automated fund release require money transmission licenses?
- **Dispute resolution** — How does the on-chain arbitration relate to existing legal dispute resolution mechanisms?

Appendix A: Project File Reference

Repository Structure

```
ai-verified-escrow/
├── compose.yml                      # Docker Compose for the verifier service
├── .env.example                     # Environment configuration template
├── run_all_variants.sh              # Script to run all sample variants end-to-end
├── contracts/
│   └── AIVerifiedEscrow.sol         # Escrow smart contract (Solidity 0.8.28)
├── verifier/
│   ├── Dockerfile                   # Verifier container definition
│   ├── requirements.txt             # Python dependencies
│   ├── main.py                      # CLI entrypoint (Click)
│   ├── contract_interface.py        # Blockchain interactions (Web3.py)
│   ├── verification_engine.py       # Pipeline orchestrator
│   ├── runners/
│   │   ├── base.py                  # Abstract runner interface
│   │   └── api_runner.py            # REST API runner (Docker + HTTP)
│   ├── evaluators/
│   │   ├── base.py                  # Abstract evaluator interface
│   │   └── openapi_evaluator.py     # OpenAI-powered API evaluator
│   └── reports/
│       └── generator.py             # JSON + Markdown + log report output
├── samples/
│   └── calculator_api/
│       ├── project.json             # Project metadata and runner config
│       ├── requirements/
│       │   └── openapi.yaml         # OpenAPI 3.0 specification
│       └── deliverables/
│           ├── complete/            # Correct implementation
│           ├── missing_endpoint/    # Missing /multiply
│           ├── buggy/               # Wrong subtraction logic
│           └── crash_on_edge_case/  # Crashes on division by zero
└── reports/                         # Generated verification reports
```

Environment Variables

Variable	Required	Description
`RPC_URL`	Yes	Ethereum JSON-RPC endpoint
`PRIVATE_KEY_CLIENT`	Yes	Client account private key
`PRIVATE_KEY_CONTRACTOR`	Yes	Contractor account private key
`PRIVATE_KEY_VERIFIER`	Yes	AI verifier account private key
`PRIVATE_KEY_ARBITER`	No	Human arbiter key (defaults to client)
`OPENAI_API_KEY`	Yes	OpenAI API key for evaluations
`OPENAI_MODEL`	No	AI model to use (default: `gpt-4o`)
`CONTRACT_ADDRESS`	Yes	Deployed contract address (set after running `deploy`)

Appendix B: Full Verification Results

B.1: `complete` — Full AI Evaluation

Score: 100/100 | **Confidence:** High | **Recommendation:** Release

Criterion	Score	Status	Details
Endpoint Completeness	100	Pass	All required endpoints are implemented and responding as expected.
Correctness	100	Pass	Endpoints return correct results for all tested standard inputs.
Error Handling	100	Pass	The API correctly handles division by zero and invalid inputs, returning appropriate error messages.
Schema Compliance	100	Pass	Responses match the expected JSON structure as defined in the OpenAPI specification.
Stability	100	Pass	The server remained responsive throughout all tests with no issues.

Tests: 16/16 passed. No issues found.

B.2: `missing_endpoint` — Full AI Evaluation

Score: 75/100 | **Confidence:** High | **Recommendation:** Fail

Criterion	Score	Status	Details
Endpoint Completeness	60	Fail	The multiplication endpoint is missing, leading to 404 errors for all multiplication tests.
Correctness	90	Pass	All implemented endpoints return correct results for standard inputs.
Error Handling	90	Pass	The API correctly handles division by zero and invalid inputs with appropriate error responses.
Schema Compliance	90	Pass	Responses match the expected JSON structure for both successful and error cases.
Stability	100	Pass	The server remained responsive throughout all tests.

Tests: 13/16 passed. 3 failures on `/multiply` (all returned 404).

Issues: Missing multiplication endpoint resulting in 404 errors.

B.3: `buggy` — Full AI Evaluation

Score: 65/100 | **Confidence:** High | **Recommendation:** Fail

Criterion	Score	Status	Details
Endpoint Completeness	100	Pass	All required endpoints are present and responding as expected.
Correctness	50	Fail	The subtraction endpoint returns incorrect results, affecting overall correctness.
Error Handling	100	Pass	Error handling for invalid inputs and division by zero is correctly implemented.
Schema Compliance	100	Pass	All responses match the expected JSON structure.
Stability	100	Pass	The server remains responsive throughout all tests.

Tests: 13/16 passed. 3 failures on `/subtract`: `10 - 3`: expected `7`, got `13` - `2 - 7`: expected `-5`, got `9` - `5.5 - 2.2`: expected `3.3`, got `7.7`

Note: `0 - 0` passed because the buggy implementation (`a + b`) produces the same result as the correct one when both operands are zero.

Issues: Subtraction endpoint returns incorrect results for all non-zero test cases.

B.4: `crash_on_edge_case` — Full AI Evaluation

Score: 85/100 | **Confidence:** High | **Recommendation:** Fail

Criterion	Score	Status	Details
Endpoint Completeness	100	Pass	All required endpoints are present and responding as expected.
Correctness	100	Pass	Endpoints return correct results for standard inputs.
Error Handling	60	Fail	The API fails to handle division by zero correctly, returning a 500 error instead of the expected 400.
Schema Compliance	100	Pass	Responses match the expected JSON structure for valid inputs.
Stability	100	Pass	The server remains responsive throughout all tests.

Tests: 15/16 passed. 1 failure on `/divide` with `b=0`: expected status 400, got status 500 with an unhandled `TypeError` stack trace.

Issues: Division by zero returns 500 instead of 400.

This paper accompanies the AI-Verified Software Delivery Escrow proof of concept, developed by Consensix Labs. The code is provided for research and educational purposes. It has not been audited for production use and should not be used with real funds without thorough security review and legal counsel.