Panda: A Decentralized Verifiable Learning Layer for EVM & Solana

May 8, 2026

`pandachain.io`

`[v0.0.1]`

Abstract

Panda introduces a portable, deterministic Python virtual machine (`PandaVM`) enabling verifiable data science smart contracts across both Ethereum and Solana. The system allows data scientists to deploy Python contracts — complete with native access to NumPy, pandas, scikit-learn, TensorFlow, PyTorch, Keras, SciPy, and XGBoost — directly on-chain with cryptographic guarantees of execution correctness. `PandaVM` achieves determinism through a six-layer enforcement strategy spanning environment-level controls, runtime overrides, import isolation, garbage collection control, and dual-execution verification. Security is enforced through an eleven-layer defense-in-depth model encompassing static analysis of forbidden modules and dangerous patterns, restricted builtins, gas metering with targeted bypass mitigations, C-extension module patching, and state validation. Two execution modes provide graduated security: the default in-process mode activates seven layers with an optional two-phase seccomp-BPF syscall filter, while a fork-based sandbox mode activates all eleven layers including memory bounds, capability dropping, and namespace isolation. A deterministic async/await primitive enables cross-block contract execution — the first on any blockchain — with two compilation modes (continuation-passing and checkpoint-resume) and a dedicated ten-layer security model addressing replay attacks, caller spoofing, code mutation, and reentrancy. The system defines three zero-knowledge proof backends with structural scaffolding for validity and fraud proofs, enabling both optimistic and ZK rollup modes on Ethereum. Cross-chain portability is achieved through a standalone Rust library with a well-defined C ABI, integrated into Geth and Solana's Agave validator. A structured Python SDK with decorators (`@contract`, `@call`, `@query`, `@proof`, `@private`) provides a familiar development experience while enforcing on-chain safety guarantees. PRC-20 and PRC-721 token standards, privacy-preserving federated learning contracts, and an AI agent interface standard complete the protocol's contribution to verifiable decentralized data science.

Contents

List of Figures

Foreword

The scientific computing industry processes trillions of data points daily. Machine learning models underpin decisions in healthcare, finance, criminal justice, and national security. Yet the infrastructure on which these models are trained, evaluated, and deployed remains entirely centralized. A model served by an API endpoint offers no guarantee that the weights behind it are the weights that were audited. A training pipeline executed in a cloud data center offers no guarantee that the dataset was not tampered with between runs. The gap between the promise of reproducible science and the reality of opaque computation has never been wider.

Blockchain technology introduced the concept of trustless verification to financial transactions. For the first time, two parties could agree on the outcome of a computation without trusting each other or any intermediary. Bitcoin proved this for simple value transfers. Ethereum generalized it to arbitrary programs. Yet even Ethereum, with its Turing-complete virtual machine, cannot express the workloads that define modern data science. Solidity was designed for token transfers and access control, not for matrix multiplication and gradient descent. The Solana Virtual Machine offers higher throughput through its register-based BPF architecture, but similarly lacks any facility for scientific computation.

This gap is not merely technical. It is philosophical. If we accept that machine learning models increasingly govern the allocation of resources, the assessment of risk, and the adjudication of disputes, then we must also accept that the correctness of these models is a matter of public interest. The European Union's AI Act, the United States' Executive Orders on AI safety, and similar regulatory efforts worldwide all point toward a future in which model auditability is not optional. Yet the tools for achieving this auditability on trustless infrastructure do not exist.

Panda was built to close this gap. The system enables data scientists to write smart contracts in Python — the language in which over ninety percent of machine learning practitioners already work — and deploy them on-chain with deterministic execution guarantees. Every invocation of a Panda contract produces a cryptographic receipt. Every receipt can be independently verified. Every model trained on-chain can be re-executed by any node in the network and produce identical results, down to the last floating-point digit.

The path to this point was not straightforward. An earlier prototype attempted to achieve similar goals using the Cosmos SDK with an embedded CPython runtime packaged as a ZIP archive. That prototype taught us that runtime restriction of Python is fundamentally brittle. Functions deleted at runtime can be recovered through introspection mechanisms, garbage collector traversal, or foreign function interfaces. Hash verification of a packaged environment does not prevent the interpreter itself from behaving non-deterministically. The lesson was clear: *compile-time removal is the only reliable form of restriction.* The current architecture reflects this lesson at every layer.

This paper presents the complete technical specification of Panda. It is written for systems engineers, cryptographers, and data scientists who wish to understand how verifiable computation can be extended to the full breadth of scientific computing. The work is ongoing, and contributions are welcome.

Introduction

2.1 Motivation

The data science pipeline as practiced today is fundamentally opaque. Training data is collected behind closed doors, model weights are stored on proprietary servers, and inference logic is hidden behind API endpoints. A user querying a machine-learning-as-a-service provider receives a prediction but no proof that the prediction was computed correctly, no evidence that the model was trained on the claimed dataset, and no assurance that the model has not been modified since its last audit.

This opacity creates concrete problems. In healthcare, a diagnostic model deployed via API may produce different results after a silent weight update. In finance, a credit scoring model may be retrained on biased data without the knowledge of regulators. In criminal justice, a recidivism prediction tool may be replaced by a cheaper alternative without disclosure. In each case, the absence of verifiability undermines the integrity of the system.

Regulatory pressure is mounting. The European Union's AI Act mandates auditability for high-risk AI systems. Executive orders from multiple governments demand transparency in algorithmic decision-making. Yet the infrastructure for achieving this transparency on trustless platforms does not exist. Current smart contract languages — Solidity for Ethereum, Rust for Solana, Move for Aptos — were designed for financial primitives. They cannot express a linear regression, let alone a neural network.

Decentralized alternatives such as federated learning and secure multi-party computation address privacy but not verifiability. A federated learning protocol can distribute training across participants, but it cannot prove that the aggregated model is correct without an on-chain settlement layer. This settlement layer requires a smart contract capable of understanding the computation it is settling. No such capability exists on any production blockchain today.

2.2 Problem Statement

Let $\mathcal{M}$ be a machine learning model, $\mathcal{D}$ a dataset, and $f_\theta(\cdot)$ a parameterized function. The problem of *verifiable data science* requires four properties to hold simultaneously:

1. **Determinism**: For identical inputs $(x, \theta, \mathcal{D})$, execution on any compliant node produces identical output y.
2. **Verifiability**: A succinct proof π exists such that any verifier can confirm $y = f_\theta(x)$ without re-execution.
3. **Portability**: The same contract code C executes identically across heterogeneous blockchain architectures.
4. **Privacy**: Optionally, model weights θ and input data x remain hidden while the correctness of y is publicly verifiable.

No existing system satisfies all four properties. Panda is designed to do so.

2.3 Contributions

This paper makes the following contributions:

1. **PandaVM**: A portable, deterministic Python virtual machine with multi-layer security, embedding CPython 3.12 and exposed via a C ABI.
2. **Dual-chain integration**: The first system to support both EVM (Ethereum) and SVM (Solana) with feature parity from a single VM library.
3. **Scientific computing stdlib**: On-chain access to NumPy, pandas, scikit-learn, TensorFlow, PyTorch, Keras, SciPy, and XGBoost with determinism guarantees.

4. **ZK proof system**: Three-backend proof architecture (RISC Zero, Halo2, SP1) with structural scaffolding; production SDK integration is planned.

5. **Privacy layer**: Encrypted contracts with ZK-verified private execution and on-chain differential privacy.

6. **Token standards**: PRC-20 (fungible) and PRC-721 (non-fungible) implemented in Python.

7. **Federated learning primitives**: On-chain secure aggregation, differential privacy, verified gradient submission, and adversarial robustness.

8. **AI agent standard (planned)**: PRC-Agent interface specification for on-chain autonomous agents with verifiable inference.

9. **Deterministic async/await**: Cross-block contract execution via continuation-passing and checkpoint-resume modes, with a dedicated ten-layer security model for replay prevention, caller preservation, code integrity, and reentrancy protection.

2.4 Paper Organization

The remainder of this paper is organized as follows. Section 2 surveys background and related work. Section 3 presents the system architecture. Section 4 details the `PandaVM`. Section 5 covers determinism enforcement. Section 6 describes the security model. Section 7 specifies the smart contract interface. Section 8 presents deterministic async/await for cross-block execution. Sections 9 and 10 cover Ethereum and Solana integration, respectively. Section 11 presents the zero-knowledge proof system. Section 12 describes the privacy layer. Section 13 details the rollup architecture. Section 14 covers token standards and on-chain primitives. Section 15 presents machine learning contracts. Section 16 discusses deployment and infrastructure. Section 17 outlines evaluation. Section 18 describes future work, and Section 19 concludes.

Background & Related Work

3.1 Smart Contract Execution Environments

The Ethereum Virtual Machine (EVM) is a stack-based, deterministic execution environment supporting smart contracts written in Solidity and Vyper [2]. The EVM guarantees determinism through its gas-metered instruction set but offers limited expressivity for scientific computation. Arithmetic is restricted to 256-bit integers, and floating-point operations are absent entirely.

The Solana Virtual Machine (SVM) uses a register-based Berkeley Packet Filter (BPF) architecture, supporting contracts written in Rust and C [3]. While offering significantly higher throughput than the EVM, the SVM similarly lacks any facility for matrix operations, statistical computation, or machine learning inference.

Alternative virtual machines — CosmWasm [23] for Cosmos, Move VM for Aptos and Sui — expand the design space but remain focused on financial primitives. None support the scientific computing libraries that define the data science ecosystem.

3.2 Verifiable Computation

Zero-knowledge proof systems enable a prover to convince a verifier that a computation was performed correctly without revealing the computation's inputs. SNARKs [6], STARKs, Plonk, and Halo2 [7] represent different tradeoffs between proof size, verification time, and trusted setup requirements. RISC Zero [15] and SP1 [16] offer general-purpose zkVMs that can verify arbitrary RISC-V programs.

Fraud proof systems, as used by optimistic rollups such as Optimism [26] and Arbitrum [27], take a complementary approach. Execution is assumed correct unless challenged. A challenger can force on-chain re-execution of a disputed step,

with the loser forfeiting a bond. This approach trades latency (the challenge window) for reduced proof generation costs.

Panda supports both paradigms: validity proofs via three ZK backends and fraud proofs via an on-chain challenge game.

3.3 Machine Learning on Blockchain

Several projects have explored the intersection of machine learning and blockchain. Ocean Protocol provides a data marketplace with compute-to-data capabilities. SingularityNET offers an AI service marketplace. Bittensor implements a decentralized network for machine learning model evaluation. Gensyn provides a distributed compute network for model training.

However, all of these systems perform computation off-chain, with only attestations or rewards settled on-chain. None provide native on-chain execution of machine learning workloads with cryptographic verification of correctness. Panda is, to our knowledge, the first system to embed a full scientific computing environment directly within a blockchain execution layer.

3.4 Privacy-Preserving Machine Learning

Federated learning [4] enables collaborative model training across distributed participants without sharing raw data. Differential privacy [5] provides mathematical guarantees that individual data points cannot be inferred from aggregate statistics. Secure multi-party computation and homomorphic encryption enable computation on encrypted data. COINSTAC [19] demonstrated an early and ambitious framework for decentralized brain imaging analysis, enabling large-scale distributed computation across neuroimaging sites without centralizing raw data. While COINSTAC focused on decentralized analysis pipelines for the neuroimaging community, it highlighted the broader need for verifiable, privacy-preserving distributed computation infrastructure — a need that Panda addresses at the smart contract layer with on-chain settlement guarantees.

Panda integrates these primitives natively. Federated learning contracts coordinate gradient aggregation on-chain. Differential privacy noise is generated deterministically from block-derived seeds. Encrypted contracts allow private model weights to be verified without disclosure.

3.5 The v0 Prototype: Lessons Learned

An earlier attempt to build this system used the Cosmos SDK with Go and an embedded CPython runtime. The Python environment was packaged as a ZIP archive with hash verification. Security relied on runtime deletion of dangerous builtins and `chmod` sealing of the packaged environment.

This approach proved insufficient. Runtime patching of Python builtins is brittle: deleted functions can be recovered via introspection mechanisms, garbage collector traversal, or foreign function interfaces. Hash verification of the environment does not prevent the interpreter itself from introducing non-determinism through hash randomization, system time access, or threading. Determinism enforcement via seed rules alone is partial at best.

The key architectural insights from `v0` were twofold. First, *compile-time removal is strictly superior to runtime restriction.* A function removed from the CPython source cannot be recovered by any contract. Second, *a portable VM library is strictly superior to a tightly-coupled chain integration.* The `v0` prototype was locked to Cosmos. The current architecture produces a standalone Rust library with a C ABI, enabling integration into any blockchain that supports C or Rust foreign function interfaces.

System Architecture

4.1 Architectural Overview

Panda is organized as a layered system with a portable core and chain-specific adapters. The central component is `PandaVM`, a Rust library that embeds CPython 3.12 [10]. This library exposes a C ABI through its FFI module, enabling integration into any host environment that supports C-compatible function calls.

The VM library is consumed by two chain adapters. On Ethereum, `panda-geth` is a fork of go-ethereum (Geth) that links the VM library via CGO. On Solana, `panda-solana` is a fork of the Agave validator that includes `PandaVM` as a native Rust dependency, registered as a builtin program.

Supporting infrastructure includes Solidity contracts on Ethereum L1 (verifier, bridge, fraud proof, and registry), a rollup sequencer and batcher, a distributed prover network, and a block explorer with React UI. The overall architecture is illustrated in Figure 1.

4.2 Design Principles

Five principles guided the architecture:

1. **Portability First**: The VM exists as a standalone library with a C ABI. Chain-specific logic is confined to thin adapter layers. Adding support for a new chain requires only writing a new adapter.
2. **Defense in Depth**: Security is enforced through eleven layers with graduated activation across two execution modes. Breaching any single layer does not compromise the system.
3. **Determinism Everywhere**: Six layers of determinism enforcement ensure identical execution across all compliant nodes, including fully deterministic gas metering.
4. **Verifiability by Default**: Every execution can produce a cryptographic receipt and, optionally, a zero-knowledge proof.
5. **Python-Native**: The system leverages the existing data science ecosystem rather than requiring practitioners to learn a new language.

4.3 Component Overview

The system comprises the following components:

- **PandaVM**: Core VM with embedded CPython, sandbox, state management, proof generation, and privacy engine.
- **Ethereum Integration**: Geth fork with CGO bridge, state adapter, gas conversion, and JSON-RPC extensions.
- **Solana Integration**: Agave validator fork with builtin program, account-based state, and thread-local VM instances.
- **L1 Contracts**: Verifier, bridge, fraud proof, and registry contracts.
- **Rollup**: Sequencer, batcher, bridge relay, and challenge game.
- **Prover Network**: Distributed proof generation.
- **SDK**: Developer SDK with decorators and type system.
- **CLI**: Developer tooling for deploy, call, query, lint, and test.

PandaVM: The Portable Python Virtual Machine

5.1 Python Runtime Integration

Stock CPython is unsuitable for blockchain execution. Several sources of

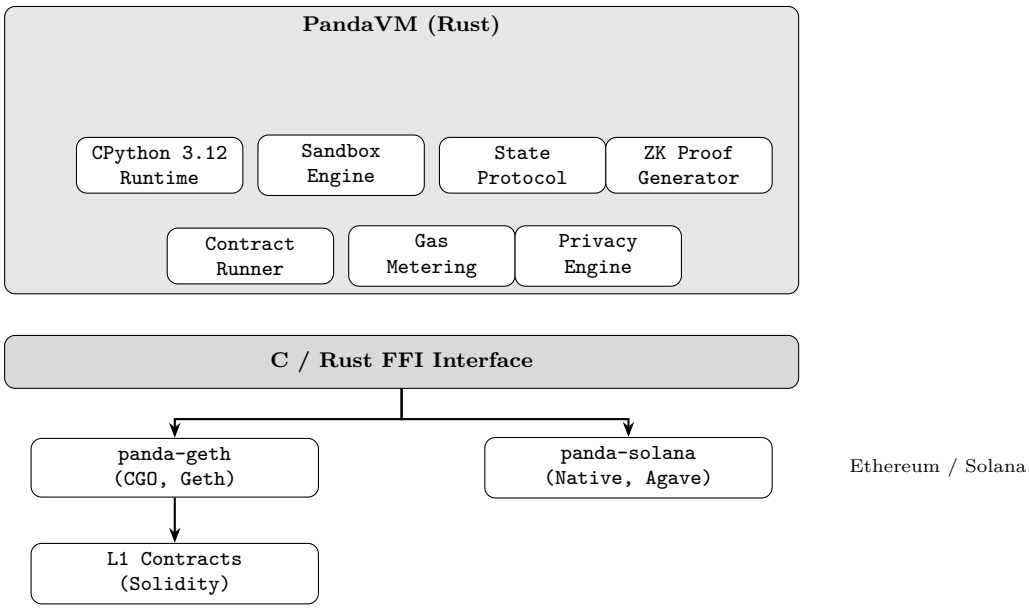

Figure 1: Panda system architecture. `PandaVM` is a portable Rust library exposing a C ABI, consumed by chain-specific adapters for Ethereum (CGO) and Solana (native Rust). On Ethereum, L1 Solidity contracts anchor the rollup.

non-determinism are baked into the interpreter at the source level. Hash randomization, enabled by default since Python 3.3, causes dictionary iteration order to vary across runs. `os.urandom()` draws from system entropy. `time.time()` returns wall-clock time. Threading primitives introduce race conditions. Native extensions require filesystem access. The import system allows loading of arbitrary modules.

The current implementation embeds stock CPython 3.12 [10] with Rust bindings [28]. All determinism and security restrictions are enforced at the Python level through multiple layers of the security model: static analysis blocks forbidden modules before execution, restricted builtins prevent dangerous calls at runtime, a custom import hook intercepts dynamic imports, and environment variables force single-threaded deterministic operation. Hash randomization is disabled, and time functions are overridden at runtime to return block-derived values.

The target architecture calls for a custom-built CPython 3.12 with compile-time source patches, which would provide strictly stronger guarantees:

1. Hash randomization disabled at the source level, not merely via environment variable.
2. `os.urandom()` redirected to a seeded CSPRNG derived from the execution context.
3. `time.time()` and `time.monotonic()` returning a virtual clock value derived from the block timestamp.
4. Threading primitives replaced with single-threaded no-ops.
5. File I/O restricted to the VM's virtual filesystem.
6. The `socket` module replaced with a compile-time stub.
7. The `subprocess` module removed entirely.
8. The import system restricted to embedded environment modules and contract modules only.

Compile-time removal is strictly superior to runtime restriction: a function removed from the CPython source cannot

be recovered by any contract, regardless of how creative the contract author may be. Until the custom build is deployed, the Python-level enforcement provides equivalent functional guarantees for all currently supported contract operations.

5.2 Scientific Computing Environment

`PandaVM` provides access to a curated set of scientific computing libraries, organized into three tiers. In the current implementation, libraries are loaded from the host Python environment and restricted via the import hook. The target architecture will package these as a sealed archive with Merkle-tree-verified integrity. The supported libraries are:

The target sealed environment will have the following properties: a SHA-256 hash verified at extraction time, per-file integrity enforced via a Merkle tree, a read-only mount preventing runtime modification, and one-time extraction cached for subsequent executions.

Within these libraries, *in-memory computation* is fully supported: array math, DataFrame operations, model training, statistical analysis, and all pure-computation APIs. *File I/O operations* are blocked: all file reader and writer functions across NumPy, pandas, and SciPy are replaced with stubs that raise errors (Layer 6). When the kernel-level filter is enabled, Phase 2 (Layer 7) additionally blocks the underlying syscalls. See the C-Extension Sandboxing subsection in the Security Model for details.

5.3 Execution Pipeline

The execution pipeline proceeds through seven stages. Stages 2 and 4 (kernel-level filter installation) are conditionally activated; the remaining stages execute unconditionally. The pre-warm and execution phases operate in separate interpreter calls sharing a common namespace.

Stage 1: Static Validation. The contract source is parsed to an AST and traversed. The scanner checks for forbidden module imports, forbidden builtin calls, and dangerous code patterns at the AST node level, making it immune to encoding bypass tricks (null bytes, Unicode normalization, string concatenation). Any violation halts execution.

Stage 2: Seccomp Phase 1 (allow-list) [conditional]. When the kernel-level filter is enabled, a Phase 1 allow-list is installed, permitting syscalls needed for shared library loading while permanently blocking process execution, debugging, and network operations with process termination. On non-Linux platforms, this stage is a no-op.

Stage 3: Pre-warm. The pre-warm phase sets determinism environment variables, derives random seeds from block context, loads the Panda SDK and standard library, pre-warms commonly used standard library and ML modules to populate C-extension caches before metering begins, and patches dangerous I/O functions on scientific computing module objects.

Stage 4: Seccomp Phase 2 (block-list) [conditional]. When the kernel-level filter is enabled, a Phase 2 block-list is installed between the pre-warm and execution calls, blocking file I/O syscalls with a permission error. Because kernel-level filters are permanent and additive, the combined effect leaves the contract with no filesystem access.

Stage 5: Sandbox Restriction. The execution phase begins. The builtins namespace is replaced with a restricted set (including a bounded `pow()`), recursion depth is capped, the import function is replaced with the restricted hook, and module entries for forbidden modules are cleared.

Stage 6: Contract Execution. The contract source is executed in the restricted namespace, the `@contract` class is instantiated, and the target method is called. A bytecode-level trace callback counts every line executed for gas metering. Three targeted mitigations address C-level gas bypass: bounded exponentiation, recursion depth limits, and an independent wall-clock timeout. In forked sand-

Table 1: Scientific computing library tiers

Tier	Libraries	Rationale
1 (Core)	NumPy, pandas, sklearn	Foundation
2 (DL)	TF, PyTorch, Keras	Neural nets
3 (Ext)	SciPy, XGBoost	Statistics

box mode (see Section 6), virtual memory is additionally capped per contract. Cross-contract calls go through a journal with reentrancy detection.

Stage 7: Result Capture. The execution result is assembled: state diff, return value, emitted events, logs, gas used, and receipt hash. State is serialized with canonical JSON (sorted keys, NaN/Infinity rejected). The full pipeline is illustrated in Figure 2.

5.4 State Management

Contracts declare their state as a nested `State` class within the contract class. Supported types include `int`, `float`, `str`, `bool`, `bytes`, `list`, `dict`, and `None`.

State access is mediated by a transparent proxy that intercepts attribute access and mutation. The proxy tracks reads and writes for diff computation, enforces type declarations (only declared fields are accessible), and materializes state from JSON on demand.

Serialization uses canonical JSON with sorted keys and NaN/Infinity rejection for deterministic encoding. Sorting keys ensures cross-platform identical byte sequences, and rejecting NaN/Infinity prevents consensus divergence from IEEE 754 equality anomalies (see Section 5.3). The Merkle root of the contract state is computed as:

$$R = \mathcal{M}(H(k_1 \| v_1), \ldots, H(k_n \| v_n))$$

where R is the state root, $\mathcal{M}$ denotes binary Merkle tree construction, H = SHA-256, k_i are lexicographically sorted state keys, and v_i are canonical JSON-encoded values.

The tree uses standard binary Merkle construction: leaf hashes are paired and each parent is computed as $H(L \| R)$, recursing until a single root remains.

5.5 Gas Metering

Panda introduces its own gas unit, the *Panda Compute Unit* (PCU). One PCU corresponds to one line of Python bytecode executed:

$$\text{gas}(C) = \sum_{i=1}^{n} \mathbb{1}[\text{event}_i = \texttt{'line'}]$$

Metering is performed via a bytecode trace callback, which fires on every line event during execution. Because single C-level operations can bypass line-level metering, targeted defenses enforce the following bounds:

$$b^e : e \leq 10{,}000, \qquad d_r \leq 200$$

where b^e is the exponentiation bound, e is the exponent, and d_r is the maximum Python recursion depth.

In forked sandbox mode, virtual memory is additionally capped per contract, catching memory bombs that bypass line-level metering entirely. An independent wall-clock timeout provides a final safety net beyond the trace-based gas metering.

Cross-contract calls incur a base cost of 100 PCU. State reads and writes are included in the line cost and do not incur additional charges.

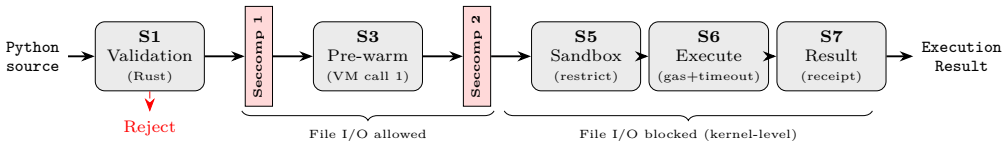

Figure 2: The `PandaVM` execution pipeline. Two kernel-level filter gates (red) provide monotonically increasing syscall restriction. File I/O is permitted during pre-warm for shared library loading; Phase 2 permanently blocks it before contract execution.

PCU values are converted to chain-native gas units at the adapter layer. The EVM conversion ratio is confirmed in the Geth integration:

$$G_{\text{evm}} = P \times 10 \tag{1}$$

where G_{evm} is the EVM gas cost and P is the Panda Compute Unit count.

The Solana conversion uses fixed gas limits per operation type rather than an explicit ratio. The target ratio is:

$$G_{\text{sol}} = P \times 5 \quad \text{(configurable)} \tag{2}$$

5.6 Cross-Contract Calls

Panda contracts can invoke methods on other deployed contracts through four primitives: `call_contract` for state-mutating invocations, `query_contract` for read-only calls, `try_call_contract` for error-safe invocations returning a `(success, result)` tuple, and `at()` for a proxy pattern that allows method calls on a remote contract as if it were a local object.

Cross-contract calls are bounded by:

$$d(c) \leq D_{\max} = 10, \qquad C_b(c) = 100 \text{ PCU} \tag{3}$$

where $d(c)$ is the call depth at invocation c, $D_{\max}$ is the maximum nesting depth, and $C_b(c)$ is the base gas cost per cross-contract call.

A reentrancy guard is maintained via an address set in the call stack. All sub-call state changes are tracked in a state journal for atomic commit or rollback.

5.7 Execution Receipts

Every contract execution produces a cryptographic receipt. The receipt hash is computed as:

$$\begin{aligned}\text{receipt} = H(&\text{code} \| \text{input\_state} \\ &\| \text{call\_data} \| \text{ctx} \| \text{output} \\ &\| \text{new\_state} \| \text{events} \| \text{logs} \| \text{gas})\end{aligned}$$

where each component is itself a hash of its respective data. Two nodes producing different receipt hashes for the same inputs indicates either non-determinism or byzantine behavior. This receipt serves as the foundation for both the fraud proof and validity proof systems.

Determinism Enforcement

6.1 The Determinism Problem in Scientific Computing

Machine learning workloads are inherently non-deterministic across several dimensions. Floating-point operation ordering differs based on compiler optimization level, BLAS implementation, and thread scheduling. Multi-threaded BLAS and LAPACK operations produce different results depending on thread interleaving. GPU kernel scheduling is non-deterministic by design. Hash table iteration order varies across Python processes due to hash randomization. Random number generation depends on entropy sources. System time dependencies create execution-environment coupling.

Table 2: Gas limits by operation type

Op	PCU	EVM	Solana
Deploy	2M	20M	10M
Call	1M	10M	5M
Query	500K	5M	2.5M

Achieving deterministic execution of scientific computing workloads requires addressing all of these sources simultaneously. Partial solutions — such as fixing the random seed alone — are insufficient.

6.2 Six-Layer Determinism Architecture

`PandaVM` enforces determinism through six independent layers, each addressing a distinct class of non-determinism sources. Figure 3 illustrates the layered architecture.

Layer 1: Compile-Time CPython Patches (planned). In the target architecture, non-determinism will be removed at the CPython source level before compilation. This is the strongest form of enforcement because it cannot be bypassed by any runtime technique. Patches target hash randomization, deterministic time functions, subprocess module removal, and adaptive bytecode specialization. The current implementation achieves equivalent functional guarantees through Layers 2–6.

Layer 2: Environment Variables. Before every execution, environment variables are configured to enforce determinism across five categories: (1) hash randomization is disabled, (2) adaptive bytecode specialization is disabled *before interpreter initialization*, (3) all numerical libraries are forced to single-threaded operation (OpenMP, MKL, OpenBLAS, NumExpr), (4) GPU access is disabled, and (5) deterministic operation modes are enabled for ML frameworks.

Disabling adaptive specialization is critical. CPython 3.11+ includes PEP 659, which dynamically rewrites bytecode during execution. Without disabling it, the trace callback fires different numbers of line events for the same code on cold versus warm runs, causing gas metering to produce different values across runs — a catastrophic consensus failure where a miner and validator compute different gas for the same transaction, causing block rejection.

Layer 3: Runtime Overrides. Beyond environment variables, certain Python-level functions must be explicitly overridden:

- `time.time()` → `ctx.block_time`
- `datetime.now()` → deterministic timestamp from block
- `random.seed()` → SHA-256(block_height‖contract_address)
- `numpy.random.seed()` → same deterministic derivation
- `torch.manual_seed()` → same deterministic derivation
- `tensorflow.random.set_seed()` → same deterministic derivation

Layer 4: Import Isolation. Module caching creates a subtle determinism threat: the first execution in a process loads a module (firing thousands of trace events = high gas), while subsequent executions find it cached (zero loading events = low gas). PandaVM addresses this with a snapshot/restore mechanism:

1. Before metered execution, the set of currently loaded modules is snapshotted.
2. The import path cache is also saved.

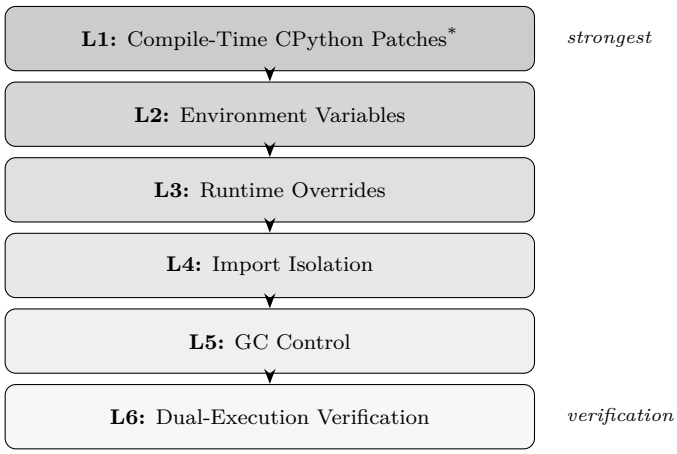

Figure 3: Six-layer determinism architecture. Each layer addresses a distinct class of non-determinism. Layer 1 (compile-time) is the target architecture (marked with *); the current implementation enforces equivalent restrictions through Layers 2–6. Layers 4–5 address the critical gas determinism problem: without import isolation and GC control, trace-level line event counts diverge between cold and warm runs, causing consensus failure.

3. Compiled regex caches from module-level code are cleared (these persist between runs and cause different trace event counts).

4. After execution (in both success and error paths), any new modules added during the metered phase are removed, the path cache is restored, and import caches are invalidated.

This ensures that every execution sees the same module loading state, producing identical gas regardless of execution history. Additionally, commonly used standard library modules with C-extension initialization caches are pre-warmed during the unmetered phase. These caches are process-global and not cleared by module restore, so pre-warming ensures they are always populated before metered execution begins.

Layer 5: Garbage Collection Control. Garbage collection finalizers execute Python code that fires trace events. If GC triggers at different points on different nodes, gas counts diverge. PandaVM disables GC during metered execution and re-enables it after execution completes (on both success and error paths). This eliminates GC-induced non-determinism in gas metering.

Layer 6: Dual-Execution Verification. When dual-execution verification is enabled, the contract is executed twice with identical inputs. The outputs are compared byte-for-byte: state, return value, events, logs, and gas consumed. Any divergence raises a non-determinism error, halting execution. With Layers 2–5 in place, gas is fully deterministic across runs, including for contracts with heavy imports and stdlib usage.

6.3 Floating-Point Determinism

Floating-point arithmetic requires special attention due to the non-associativity of IEEE 754 [9] operations. The same mathematical expression can produce different results depending on the order in which additions and multiplications are performed. Multi-threaded BLAS implementations routinely reorder operations for performance, producing non-deterministic results.

`PandaVM` addresses this through several mechanisms. All floating-point results in the `panda.ml` standard library

are rounded to 10 decimal places using IEEE 754 round-half-even (banker's rounding):

$$\text{round}_{10}(x) = \text{round\_half\_even}(x \cdot 10^{10}) \cdot 10^{-10}$$

Single-threaded BLAS ensures consistent operation ordering. Execution is restricted to CPU only, as GPU floating-point ordering varies by device and driver version. The pure-Python fallback in `panda.ml` uses deterministic Gaussian elimination.

Formally, for any contract C and inputs (s, d, ctx):

$$\forall \text{node}_i, \text{node}_j : \\ \text{exec}(C, s, d, \text{ctx})_i = \text{exec}(C, s, d, \text{ctx})_j \quad (4)$$

provided both nodes run compliant `PandaVM` implementations.

6.4 State Serialization Determinism

Beyond floating-point computation, deterministic state serialization is critical for consensus. Two mechanisms ensure canonical state encoding:

Canonical JSON. All state serialization uses `json.dumps(sort_keys=True, allow_nan=False)`, producing a canonical byte sequence regardless of Python dictionary insertion order. While CPython 3.7+ guarantees insertion-order iteration, different execution paths may produce dicts with identical keys but different insertion orders. Sorting keys before encoding eliminates this source of non-determinism.

NaN and Infinity rejection. The serializer rejects NaN and Infinity values, raising an error if any state field contains these values. This is critical because NaN violates IEEE 754 reflexivity (`NaN != NaN`), meaning two nodes that both compute NaN would agree on the serialized value but disagree on equality comparisons in subsequent contract logic. Rejecting NaN at the serialization boundary forces contracts to handle edge cases explicitly rather than allowing silent consensus divergence.

6.5 Empirical Validation

The determinism guarantee is validated across float-heavy arithmetic operations, dictionary ordering under hash randomization, large integer arithmetic (Fibonacci, factorial), ML model training (scikit-learn `LinearRegression`), random sequence generation with fixed seeds, and receipt hash consistency across multiple runs.

Gas determinism is specifically validated by executing contracts with standard library and ML imports across multiple consecutive runs, asserting identical gas consumption on every run. Test categories include: metered standard library imports, ML library imports, cross-contract interleaved imports where standalone gas matches interleaved gas, and import-heavy contracts producing identical gas *and* state across multiple runs. NaN and Infinity injection is validated to produce clean errors with descriptive tracebacks rather than crash or silent corruption.

Security Model

7.1 Threat Model

The threat model assumes adversarial contract authors with deep knowledge of CPython internals, C extension interfaces, and Linux syscall conventions. Attackers may attempt any of the following: escaping the sandbox to access the host filesystem, network, or process space; causing non-deterministic execution to exploit consensus divergence; exhausting computational resources (CPU, memory, disk); accessing state belonging to other contracts; corrupting their own state through undeclared fields; and extracting information from the host environment.

Several attack categories deserve

particular attention due to the unique challenges of sandboxing Python with scientific computing libraries:

- **C-extension bypass**: NumPy, pandas, and SciPy implement I/O functions in C, which call OS-level `open()` directly, bypassing Python-level `builtins.open` restrictions. A contract calling `np.save('/tmp/data.npy', secret)` would succeed unless both the Python call path and the kernel syscall path are blocked.

- **Pre-warm exploitation**: During ML library initialization, file I/O must be permitted for `.so` loading. An attacker may craft a contract that exploits this timing window through a supply-chain attack on a bundled library.

- **Gas metering bypass**: Expressions such as `[0]*10**9`, `"A"*10**9`, or `10**(10**8)` are single Python line events that execute entirely in C between gas meter checks. These are mitigated by bounded exponentiation, recursion limits, a wall-clock timeout, and — in forked sandbox mode — memory caps.

- **State serialization attacks**: NaN and Infinity values violate IEEE 754 equality (`NaN != NaN`), causing nodes that both compute NaN to disagree on state comparisons. Circular references and oversized state blobs consume unbounded storage.

- **Constructor re-entry**: An attacker calls `__constructor__` on an already-deployed contract to re-initialize state and steal assets.

The security model does not protect against vulnerabilities in the contract's own logic (e.g., reentrancy in application-level token transfers) but does enforce isolation, termination, confinement, integrity, and determinism at the VM level.

7.2 Defense-in-Depth Security Model

Security is enforced through an eleven-layer architecture, illustrated in Figure 4. The system provides two execution modes with graduated security. In *in-process mode* (default), seven layers are always active (L1, L3–L6, L10–L11), with an eighth (L7, seccomp-BPF) available via configuration. In *forked sandbox mode*, all eleven layers are active, including memory bounds (L5), capability dropping (L8), and namespace isolation (L9). Layer 2 (custom CPython build) is planned. Each layer addresses a distinct attack surface, and multiple layers defend against each attack category.

Layer 1: Static Code Analysis (active). Before execution, the contract source is parsed to an AST and traversed. The scanner checks for forbidden module imports, forbidden builtin calls, dangerous attribute access patterns, and forbidden function definitions. AST-level analysis cannot be bypassed by encoding tricks (null bytes, Unicode normalization, string concatenation) that defeat string-matching approaches.

Layer 2: CPython Integration (planned). The current implementation embeds stock CPython 3.12, with dangerous modules blocked at the static analysis (Layer 1) and runtime import hook (Layer 4) levels. The target architecture will remove dangerous modules at the source compilation level, which is strictly superior because removed code cannot be recovered through any reflection mechanism.

Layer 3: Restricted Builtins (active). Each contract executes with a restricted builtins namespace. Dangerous evaluation and I/O functions raise errors. The import function is replaced with a restricted hook.

Layer 4: Runtime Import Hook (active). A restricted import hook intercepts all dynamic import attempts. Even if a contract bypasses static analysis through dynamic string construction, the runtime hook catches the forbidden import. The hook includes submodule match-

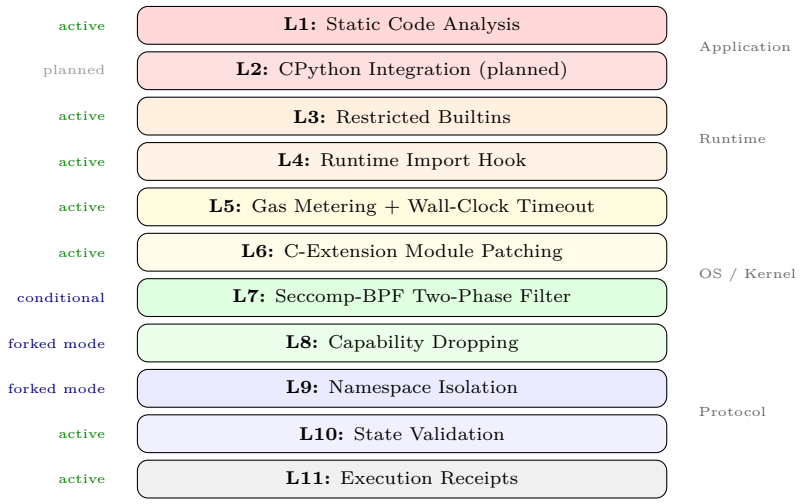

Figure 4: Eleven-layer defense-in-depth security model. Layers span from application-level static analysis (L1) through runtime enforcement (L3–L6), OS-level sandboxing (L7 conditional; L8–L9 active in forked sandbox mode), and protocol-level verification (L10–L11). Activation status is shown at left.

ing, catching imports such as `os.path` even when `os` alone is blocked.

Layer 5: Gas Metering + Wall-Clock Timeout (active). A bytecode-level trace callback counts every line executed. When the PCU limit is reached, an out-of-gas exception is raised. Gas metering starts *after* the pre-warm phase — SDK and library loading is not metered, as contracts would otherwise exhaust their gas budget on framework setup. Because CPython can silently absorb trace callback exceptions, a post-execution gas check verifies the counter. GC is disabled during metered execution to eliminate non-deterministic trace events (see Section 5). A module snapshot/restore mechanism ensures identical loading state across runs. Two targeted mitigations address C-level gas bypass: bounded exponentiation and recursion depth limits. An independent wall-clock timeout provides a final safety net. In forked sandbox mode, virtual memory is additionally capped per contract, catching memory bombs. Memory caps are not applied in-process because they would limit the entire host process address space, preventing ML library loading.

Layer 6: C-Extension Module Patching (active). Scientific computing libraries implement I/O functions in C, which bypass Python-level builtins restrictions by calling OS-level syscalls directly. After ML libraries are pre-warmed but before the contract executes, PandaVM patches dangerous functions on the already-imported module objects. For NumPy, all file save/load operations and FFI escape functions are blocked. For pandas, all file reader and writer methods are blocked. For SciPy, matrix I/O functions are blocked. Pure computation operations (array math, DataFrame operations, statistical aggregation, all arithmetic and slicing) remain fully functional.

Layer 7: Two-Phase Seccomp-BPF Syscall Filter (conditional). A two-phase kernel-level syscall filter uses Linux's seccomp-BPF mechanism [18]. Phase 1 is an allow-list permitting syscalls needed for shared library loading while permanently blocking process execution, debugging, and networking. Phase 2 is a block-list removing file I/O syscalls. Because seccomp filters are permanent and additive, the combined effect is monotonically increasing restriction that cannot be circumvented by any userspace code. In in-process mode, the two-phase filter is activated via configuration. In forked sandbox mode,

a single-phase strict filter is installed unconditionally. On non-Linux platforms, this layer is a no-op. The two-phase architecture is described in detail below.

Layer 8: Linux Capability Dropping (forked mode). All Linux capabilities are dropped in the forked child process. Even if the sandbox is breached at a higher layer, the process cannot escalate privileges.

Layer 9: Namespace Isolation (forked mode). Isolated PID, network, mount, user, and IPC namespaces are created for the forked child process. The contract sees only its own process, has no network access, and operates on a read-only filesystem mount.

Layer 10: State Validation (active). A transparent state proxy enforces type declarations at the application level. Only fields declared in the `State` class are accessible. Attempts to read or write undeclared fields raise an error, preventing state corruption and field injection. Canonical JSON serialization rejects NaN and Infinity values that would violate consensus (see Section 5.3). A deployment sentinel prevents constructor re-entry on already-deployed contracts.

Layer 11: Execution Receipts (active). The cryptographic receipt produced after every execution serves as a final verification layer. Receipt comparison across nodes detects any divergence caused by byzantine behavior or undetected nondeterminism.

7.3 Attack Surface Analysis

7.4 Formal Security Properties

The security model is designed to guarantee five properties:

Property 1 (Isolation) *No contract execution can observe or modify state outside its declared `State` class.*

Property 2 (Termination) *Every contract execution terminates within bounded gas and wall-clock time.*

Property 3 (Confinement) *No contract execution can access the filesystem, network, or host processes.*

Property 4 (Determinism) *For identical inputs, every compliant node produces identical outputs (follows from Section 5).*

Property 5 (Integrity) *No contract execution can modify its own bytecode, the VM's internal state, or the execution pipeline's security enforcement mechanisms during execution.*

Property 5 is enforced by: restricted builtins preventing `exec`/`compile`/`eval` (L3), the `sys` module being forbidden (L1/L4, preventing trace callback removal which would disable gas metering), and the contract executing in a restricted namespace separate from the harness globals. When seccomp is enabled, it additionally blocks debugging syscalls (L7, preventing self-debugging).

7.5 Two-Phase Seccomp-BPF Architecture

Running scientific computing libraries on a blockchain presents a unique sandboxing challenge: ML libraries require filesystem access during initialization (to load shared object files via `dlopen()`), but contract execution must be confined from the filesystem entirely. A single seccomp filter cannot satisfy both requirements.

PandaVM solves this with filter stacking. Seccomp-BPF filters are permanent and additive: once installed, a filter cannot be removed or weakened, and installing a second filter creates a logical AND — every syscall must pass *all* installed filters.

Phase 1 (Allow-list): Installed before ML pre-warming. Permits syscalls needed for shared library loading (file I/O, memory mapping). Permanently blocks process execution, process forking (except thread creation), debugging, all socket operations, and signal delivery. Rejected syscalls result in immediate process termination, not a recoverable error.

Table 3: Attack categories and corresponding defense layers

Attack Category	Example	Defense Layers
Sandbox escape	`import os; os.system()`	L1 (static) + L3 (builtins) + L4 (hook)
Code injection	`eval("__import__('os')")`	L1 + L3 + L4
Resource exhaustion	`while True: pass`	L5 (gas metering + wall-clock timeout)
Memory bomb	`[0] * 10**9`	L5 (bounded exponentiation + wall-clock timeout; memory limits in forked mode)
C-extension I/O bypass	`np.save('/tmp/d.npy', s)`	L6 (module patching); L7 (seccomp, when enabled)
Gas metering bypass	`10**(10**8)`	L5 (bounded exponentiation + wall-clock timeout)
Non-determinism	`time.time()`	L2 (CPython patch) + L3 (override)
State serialization	`float('nan')` in state	L10 (canonical JSON serialization)
State corruption	Undeclared field access	L10 (state proxy)
Constructor re-entry	Call `__constructor__` post-deploy	L10 (deployed sentinel)
Information leakage	`os.environ`	L1 + L4; L7 (when enabled)
Privilege escalation	`ctypes.cdll`	L1 + L4 + L8 (capabilities)
Network access	`import socket`	L1 + L4; L7 (when enabled)
Object introspection	`__subclasses__()` chain	L1 (dangerous patterns) + L3 (restricted builtins)
Cross-contract attack	Reentrancy	Depth limit 10 + address stack + journal

Phase 2 (Block-list): Installed after pre-warming, before contract execution. Blocks all file I/O syscalls — file open/create, directory operations, file metadata access, symlink traversal, and their variants. Rejected syscalls return a permission error rather than terminating the process, as these represent expected behavior (e.g., a library attempting to open a config file).

The combined effective syscall set is:

$$S_{\text{eff}} = S_{\text{Phase1}} \setminus S_{\text{Phase2}}$$

The effective syscall set after both phases leaves no filesystem access. This architecture provides three desirable properties:

1. **Monotonically increasing restriction**: security only gets tighter over time; no phase can weaken a previous phase's guarantees.

2. **No timing window**: Phase 2 is installed between two VM invocations, with no Python code executing during the transition.

3. **Kernel enforcement**: cannot be bypassed by any userspace code, including CPython internals, C extensions, or JIT-compiled code.

The syscall flow is illustrated in Figure 5.

7.6 C-Extension Sandboxing

A fundamental challenge in sandboxing Python for blockchain use is that the most valuable scientific computing libraries (NumPy, pandas, SciPy, TensorFlow, PyTorch) are implemented primarily in C/C++/Fortran with Python wrappers. These C extensions bypass Python-level security restrictions entirely.

Consider the standard Python sandbox approach of replacing the built-in file open function with a stub that raises an error. This prevents pure Python code from opening files. However, C-extension I/O operations (e.g., NumPy's array save or pandas' CSV export) call the OS-level file open syscall directly, bypassing the Python-level restriction entirely.

PandaVM addresses this with a two-layer defense:

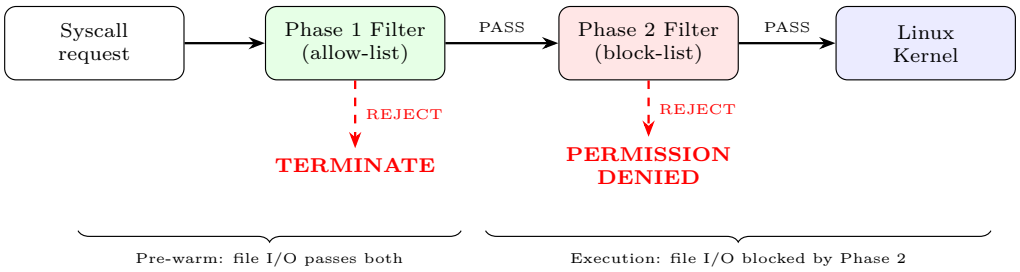

Figure 5: Two-phase seccomp-BPF filter architecture. During pre-warm, file I/O syscalls pass both filters (Phase 2 not yet installed). During contract execution, Phase 2 blocks all file I/O. Dangerous syscalls (process execution, debugging, sockets) are terminated by Phase 1 in both phases.

Layer 1 (Python-level, L6): After pre-warming ML libraries, PandaVM patches dangerous functions on the module objects themselves, replacing them with stubs that raise errors. This catches the Python-level call path.

Layer 2 (Kernel-level, L7): When seccomp is enabled, Phase 2 blocks file I/O syscalls at the kernel level. Even if an attacker bypasses the Python-level patch (e.g., by caching a C function pointer during import before patching occurs), the kernel blocks the underlying syscall. Without seccomp enabled, C-extension I/O defense relies on the Python-level module patching (L6) alone.

The patched functions are listed in Layer 6 above. All pure computation operations remain fully functional — array construction, linear algebra, dataframe manipulation, aggregation, arithmetic, indexing, and slicing. This enables full scientific computing capability while blocking all I/O escape paths.

7.7 Execution Modes

PandaVM supports two execution modes, providing graduated security guarantees:

In-Process Mode (default). Contracts execute within the host process (Geth or Agave validator). This mode is compatible with both chain integrations, incurs minimal overhead, and activates seven security layers unconditionally (L1, L3–L6, L10–L11). Seccomp-BPF (L7) is available as an opt-in but defaults to off. Address space memory limits are not applied because they would restrict the entire host process, preventing ML library loading during pre-warm.

Forked Sandbox Mode (opt-in). When the forked sandbox feature is enabled, contracts can execute in a forked child process with full OS-level isolation. The child undergoes a six-step hardening sequence: (1) Linux namespace isolation (PID, network, mount, user, IPC), (2) capability dropping (all Linux capabilities), (3) platform sandbox profile, (4) inherited file descriptor cleanup, (5) resource limits (address space, CPU time, open files, file size, process count), and (6) seccomp-BPF installation (last, as it is permanent). The parent and child communicate via a serialized result pipe. This mode activates all eleven security layers but trades execution speed for stronger isolation.

The choice between modes reflects a fundamental tension in sandboxing scientific computing: address space limits are the only defense against C-level memory bombs in in-process mode, but applying them would break ML library initialization. Forked mode resolves this by giving each contract execution a fresh address space where memory limits are safe to enforce.

7.8 Security Model Comparison

Table 4 compares PandaVM's security model to other blockchain execution environments. PandaVM's model is necessarily more complex because it runs an interpreted language with C extensions — a fundamentally harder sandboxing problem than compiled bytecode.

Smart Contract Interface

8.1 Contract Structure

A Panda smart contract is a Python class decorated with `@contract`. The class contains a nested `State` class declaring typed fields with default values, and methods decorated with role-specific decorators. The minimal contract structure is:

```python
from panda import contract, constructor,
    call, query, event, private, proof

@contract
class MyContract:
    class State:
        field1: int = 0
        field2: str = ""

    @constructor
    def init(self, ctx, param1):
        self.state.field1 = param1

    @call
    def mutate(self, ctx, arg1):
        self.state.field1 = arg1
        self.emit(event.Updated(
            value=arg1, sender=ctx.sender
        ))

    @query
    def read(self, arg1) -> int:
        return self.state.field1
```

Listing 1: Panda contract structure

8.2 Execution Context

Every state-mutating method receives an execution context object `ctx` providing consensus-derived information:

- `ctx.sender` — caller address (hex string)
- `ctx.block_height` — current block number
- `ctx.block_time` — block timestamp (Unix seconds)
- `ctx.contract_address` — this contract's address
- `ctx.chain_id` — `"panda-evm"` or `"panda-solana"`
- `ctx.is_deploy` — `True` during constructor
- `ctx.call_depth` — cross-contract nesting depth
- `ctx.origin_sender` — original transaction sender

8.3 Decorator Semantics

The `@query` decorator marks methods as read-only. Any state mutations performed within a query method are discarded after execution, ensuring that queries cannot alter on-chain state. The `@private` decorator enables encrypted execution, adding encryption overhead to the base gas cost. The `@proof` decorator marks methods for proof generation with `type="validity"` or `type="fraud"` modes. Proof generation itself is performed as a separate step by the prover network, not inline during contract execution.

8.4 Event System

Contracts emit events via `self.emit()`, passing a named event object. Events are serialized as JSON with a name and data payload, included in the execution result, and indexed on-chain for querying through Ethereum logs or Solana program logs.

Deterministic Async/Await

9.1 Related Work: Cross-Block Execution

No existing blockchain supports native `async/await` semantics within smart con-

Table 4: Security model comparison across blockchain VMs

Feature	EVM	SolanaVM	CosmWasm	PandaVM
Language	Solidity (compiled)	Rust → eBPF	Rust → Wasm	Python (interpreted)
Sandbox type	Stack machine	eBPF verifier	Wasm sandbox	11-layer defense
Kernel isolation	No	No	No	Seccomp-BPF (conditional)
C extension support	N/A	N/A	N/A	Yes (with patching)
Determinism	Inherent (no floats)	Inherent (no floats)	Inherent (Wasm)	6-layer enforcement
Gas metering	Opcode-level	Compute units	Wasm instruction	Line-level + wall-clock + bypass mitigations

Table 5: Decorator properties

Decorator	Mutation	Tx Req.	Gas
`@constructor`	Yes	Yes	Deploy
`@call`	Yes	Yes	Call
`@query`	No	No	Query
`@private`	Yes	Yes	Call+
`@proof`	Yes	Yes	Call+

tracts. Several systems approximate cross-block execution but fall short of the programming model Panda provides:

- **Near Protocol Promises** [22]: Cross-contract async calls via `Promise::then()`, but all promises resolve within a single block. Multi-block suspension is not supported.
- **Cosmos IBC Callbacks**: Cross-chain async at the module level, but not within individual contract code. Developers cannot write `await` inside a contract method.
- **Ethereum Account Abstraction (EIP-7702)**: Enables multi-step transaction bundling but not mid-execution suspension or cross-block continuations.
- **Solana Durable Nonces**: Allow deferred transaction submission but not mid-execution suspension. The program cannot "pause" and resume.
- **Cartesi** [24]: Off-chain Linux VM with on-chain verification, but contracts do not use `async`/`await` syntax; the execution model is batch-oriented.
- **Aztec's Noir** [25]: Private async (proving), not execution async. The contract itself runs synchronously.

Panda's differentiator is native Python `async`/`await` syntax that compiles to deterministic cross-block execution with a formal security model. The developer writes natural Python; the compiler handles CPS/hibernation, sender preservation, replay prevention, and deterministic serialization.

9.2 Motivation

Every production blockchain restricts smart contract execution to a single transaction. An Ethereum transaction that exceeds the block gas limit fails. A Solana instruction that exceeds 1.4 million compute units is aborted. Long-running computations — multi-epoch ML training, governance timelocks, vesting schedules, multi-stage data pipelines — must be manually decomposed into separate transactions

with ad-hoc state management. This decomposition is error-prone, exposes intermediate state to manipulation, and forces developers to implement their own continuation logic.

Panda introduces the first deterministic `async/await` primitive for smart contracts. Unlike Python's cooperative multitasking (which yields within a single event loop), Panda's `await` is a *block boundary*: execution suspends at the `await`, the current transaction completes, and a deterministic timer schedules resumption in a future block. The developer writes natural Python `async def` methods; the compiler handles all cross-block mechanics, sender preservation, replay prevention, and deterministic serialization.

9.3 Programming Model

The `def` versus `async def` distinction declares execution boundaries at the source level:

- `def _helper(self, x)` — synchronous, executes in the same block as the caller.
- `async def _helper(self, ctx, x)` — asynchronous, each `await` creates a block boundary.
- `await sleep(blocks=N)` — explicit N-block delay (built-in, no import needed).
- `result = await self._compute(ctx, data)` — call async helper, resume with return value.

Await targets are restricted at load time via AST validation. Only two patterns are permitted: `await sleep(blocks=N)` (or `await sleep(until_block=N)`) and `await self.method(...)`. Arbitrary awaitables, external coroutines, and `asyncio` primitives are rejected statically — `asyncio` is among the forbidden modules (see Section 6). This static restriction eliminates an entire class of non-determinism: the set of possible execution paths is known at deploy time.

```python
from panda import contract, call, query, sleep

@contract
class DataPipeline:
    class State:
        raw_data: list = []
        result: int = 0
        status: str = "idle"

    def _validate(self, data):
        """Sync helper -- same block."""
        return [x for x in data if x >= 0]

    async def _compute(self, ctx, data):
        """Async helper -- next block."""
        total = sum(data)
        await sleep(blocks=5)
        return total

    @call
    async def run_pipeline(self, ctx):
        cleaned = self._validate(
            self.state.raw_data)
        total = await self._compute(ctx, cleaned)
        self.state.result = total * 2
        self.state.status = "done"
```

Listing 2: Async contract with cross-block execution

9.4 CPS Mode (Default)

In CPS (continuation-passing style) mode, the AST transformer splits each `async def` method into multiple *phase functions* at `await` boundaries. For a method with k `await` expressions, the transformer generates $k + 1$ phase functions:

- **Phase 0**: Executes code before the first `await`, captures local variables as timer arguments, and schedules Phase 1 via a deterministic timer.
- **Phase** i: Executes code between `await` $i - 1$ and `await` i, scheduling Phase $i + 1$.
- **Phase** k: Executes code after the last `await` and pops the continuation stack to resume the caller.

Nested async calls are managed via a continuation stack stored in contract state. Each frame records the caller's next phase method, captured arguments, and (optionally) a return value target. When an async helper completes, it pops the top

continuation and invokes the caller's next phase, passing the return value. Stack growth is bounded by a hard depth limit of 10.

Circular dependency detection via depth-first search at load time prevents infinite recursion.

Sender identity is preserved across block boundaries via the `_caller` mechanism. At Phase 0, the original `ctx.sender` is captured into `_caller`. All subsequent phases receive `_caller` as a parameter through timer arguments. This is critical because timer-fired callbacks have the timer system as `ctx.sender`, not the original user. Without `_caller` capture, post-await authorization checks would be bypassed.

9.5 Hibernate Mode

Hibernate mode, selected via a contract-level metadata declaration, takes a different approach. Instead of splitting into separate functions, the hibernate transformer keeps the original function body and inserts checkpoint dispatch logic. Each `await` becomes a checkpoint: before suspension, all local variables are serialized into a hibernation state field; on resumption, locals are restored and execution jumps to the appropriate checkpoint.

Hibernate mode enforces three additional constraints:

1. **Type whitelist**: Only `int`, `float`, `str`, `bool`, `list`, `dict`, and `None` are allowed in hibernation state. Custom objects with `__reduce__` or `__getstate__` hooks are rejected, preventing deserialization attacks.

2. **Size limit**: Serialized locals are bounded by $|\text{json}(\texttt{locals})| \leq 65{,}536$ bytes, enforced via canonical JSON serialization. This bounds on-chain storage consumption.

3. **Code integrity**: A hash $h = \text{SHA-256}(\texttt{source})$ is computed at load time and stored with each checkpoint. On resumption, the hash is recomputed and verified: $h_{\text{resume}} = h_{\text{checkpoint}}$. A mismatch — indicating the contract was upgraded between phases — raises a hard error, preventing resumed code from operating on stale assumptions.

9.6 Mode Comparison

CPS mode is the default and imposes fewer constraints: local variables are passed as timer arguments rather than serialized, eliminating type restrictions. Hibernate mode is better suited for methods with many local variables or complex control flow, as it preserves the original code structure and detects contract upgrades between phases. Both modes produce identical observable behavior: the same final state, events, and gas consumption for identical inputs. The choice is a developer-facing ergonomic decision with no consensus impact.

9.7 Cross-Block Security Model

Cross-block execution introduces six threat vectors absent from single-transaction smart contracts. Each is addressed by a dedicated defense mechanism.

Threat 1: Replay Attacks. After a hibernated method completes, an attacker re-submits the resume transaction to re-execute the post-await code. Defense: one-shot clear — hibernation state is deleted *before* the resumed code executes. A replayed resume finds empty state and starts from checkpoint 0, which is idempotent.

Threat 2: Caller Identity Spoofing. Timer callbacks have the timer system as `ctx.sender`, not the original user. Post-await code that checks `ctx.sender` for authorization would be bypassed. Defense: `_caller` is captured at Phase 0 from `ctx.sender` and propagated through all subsequent phases via timer arguments.

Threat 3: Code Mutation Between Phases. A contract upgrade be-

Table 6: CPS vs. Hibernate mode tradeoffs

Property	CPS Mode	Hibernate Mode
State overhead	Continuation stack only	Full local variable snapshot
Code size	$k + 1$ phase functions generated	Single function with dispatch
Resume cost	Direct function call	Dictionary lookup + restore
Max locals	Unlimited (passed as params)	64 KB serialized (JSON)
Type restriction	None	int/float/str/bool/list/dict/None
Debugging	Phase functions visible in stack	Original code structure preserved
Code mutation detection	No	SHA-256 hash verified on resume

tween Phase 0 and Phase 1 causes resumed code to operate on different logic than what was suspended. Defense: SHA-256 code hash computed at load time, stored in hibernation state, verified at resume. Mismatch raises a hard error (hibernate mode only; CPS mode is stateless across phases).

Threat 4: Reentrancy During Hibernation. While method A is hibernated, a new call to method A arrives. It could corrupt the hibernation state or re-execute the pre-await code. Defense: per-method reentrancy guard — if the hibernation state contains an entry for the target method and the call is user-initiated (not a timer callback), execution is rejected.

Threat 5: Serialization Injection. A contract stores a custom object with a `__reduce__` hook in a local variable. During hibernation, this object is serialized and later deserialized, triggering arbitrary code. Defense: type whitelist validation rejects all types not in {int, float, str, bool, list, dict, None} before serialization. The 64 KB size limit provides a secondary bound.

Threat 6: Composition Depth Bomb. Deeply nested `await self.a()` → `await self.b()` → ... chains exhaust the continuation stack or produce unbounded state growth. Defense: hard depth limit of 10 on the continuation stack, and circular dependency detection via depth-first search at load time.

9.8 Determinism Properties

Cross-block execution introduces four additional determinism requirements beyond those in Section 5.

Property 6 (Phase Determinism) *For identical inputs (state, block_height, sender, arguments), each phase function produces identical outputs (new state, events, gas used, timer requests). This is verified by running each phase twice and comparing byte-for-byte.*

Property 7 (Timer Determinism) *Timer IDs are derived deterministically from the contract address and an execution nonce. Timer scheduling is part of the phase output. Two validators processing the same block schedule identical timers with identical fire-at-block targets.*

Property 8 (Serialization Determinism) *Hibernation state uses canonical JSON serialization (see Section 5.4). Dictionary key ordering is canonical. NaN and Infinity values are rejected. Two validators serializing the same locals produce identical byte sequences.*

Property 9 (Per-Phase Gas Determinism) *Each phase has its own gas budget. The bytecode-level trace callback counts line events independently per phase. Adaptive specialization disabling and module snapshot/restore (see Section 5.2) ensure identical gas across cold and warm runs within each phase.*

Ethereum Integration

10.1 Architecture

The Ethereum integration is implemented as a fork of go-ethereum (Geth). The PandaVM Rust library is compiled as

Table 7: Ten-layer defense-in-depth model for cross-block execution

Layer	Time	Mechanism	Threat
Await target whitelist	Load	AST validation: only `sleep()`/`self.method()`	Arbitrary async I/O
Circular dependency DFS	Load	Static call graph analysis	Infinite recursion
`_caller` reservation	Load	Parameter name reserved at AST level	Sender shadowing
Code hash computation	Load	SHA-256 of contract source	Code mutation
`_caller` capture	Runtime	Original sender saved at Phase 0	Caller spoofing
Continuation depth limit	Runtime	Max 10 levels on continuation stack	Stack exhaustion
Reentrancy guard	Runtime	Per-method hibernation check	State corruption
One-shot clear	Runtime	Hibernation state deleted before resume	Replay attacks
Type whitelist	Runtime	Only JSON-safe primitives serialized	Deserialization injection
Size limit	Runtime	64 KB max serialized locals	DoS / memory exhaustion

a shared library and linked into Geth via CGO. A minimal set of upstream Geth files are patched to support dual-VM coexistence: Solidity contracts continue to execute on the standard EVM, while Python contracts are routed to PandaVM.

10.2 Transaction Routing

When a transaction arrives at the state transition function, the system checks whether the target address is a Panda contract by inspecting a flag in the account's state. If the flag is set, execution is routed to PandaVM through the CGO bridge. Otherwise, the standard EVM execution path is followed. This dual-VM routing is illustrated in Figure 7.

10.3 Fork Activation

Panda functionality is activated via a time-based fork, following the post-merge Geth convention. Before the activation timestamp, all Panda-related transaction types are rejected. After activation, both EVM and PandaVM contracts coexist on the same chain.

10.4 State Mapping

Panda contract state is stored within the Ethereum state trie. Each Panda contract account includes the standard Ethereum fields (nonce, balance, storageRoot, codeHash) plus a Panda contract type flag. Contract state fields are mapped to storage slots:

$$s_k = \kappa(\pi \| f), \qquad s_v = \mathcal{J}(v_f)$$

where s_k is the storage slot key, $\kappa =$ `keccak256`, $\pi =$ `"panda.state."` is the namespace prefix, f is the field name, s_v is the storage value, and $\mathcal{J}(v_f)$ is the canonical JSON encoding of field value v_f.

10.5 Gas Conversion

The Panda Compute Unit to EVM gas conversion is:

$$G_{\text{evm}} = P \times 10 \tag{5}$$

where G_{evm} is the EVM gas cost and P is the Panda Compute Unit count.

10.6 Deployer Precompile

A deployer precompile is registered at a reserved address. The precompile accepts Python source code as calldata, validates it through the static analyzer, and returns a contract address derived via a CREATE2-style computation.

10.7 JSON-RPC Extensions

Nine custom JSON-RPC methods extend the standard Ethereum API: `panda_deployContract`, `panda_callContract`, `panda_queryContract`, `panda_getContractState`, `panda_getContractCode`, `panda_lintContract`, `panda_estimateGas`, `panda_getProof`,

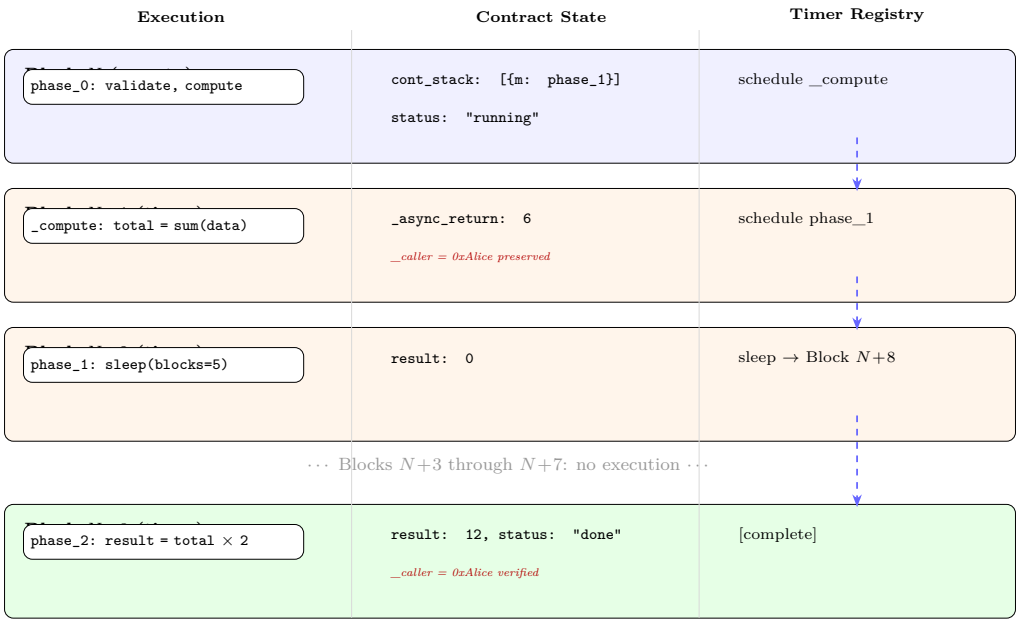

Figure 6: Async contract execution lifetime across blocks. Blue indicates user-initiated execution, orange indicates timer-fired callbacks, and green indicates completion. Dashed arrows represent timer scheduling. The `_caller` identity is preserved across all block boundaries. A five-block sleep (Blocks N+3 through N+7) demonstrates the delay primitive.

and `panda_verifyProof`. These methods provide direct programmatic access to Panda functionality without requiring users to construct raw transactions.

Solana Integration

11.1 Architecture

The Solana integration is implemented as a fork of the Agave validator. Unlike the Ethereum integration, which requires a CGO bridge, the Solana integration includes PandaVM as a native Rust dependency. The VM is registered as a builtin program, analogous to the System Program or Token Program, resulting in zero FFI overhead.

11.2 Program ID

The Panda loader program is registered at a well-known program ID, analogous to the System Program or Token Program addresses.

11.3 Instruction Set

The loader program supports four instructions, identified by a single-byte discriminant:

11.4 Account Model

The account layout varies by instruction type:

- **Deploy** (4 accounts): Signer, Program Account, State PDA, System Program
- **Call** (3 accounts): Signer, Program Account, State PDA
- **Query** (2 accounts): Program Account, State PDA (no signer required)
- **Upgrade** (2 accounts): Authority (signer), Program Account

In the current implementation, accounts are passed in by the caller at transaction time. The Solana processor validates account ownership and structure but

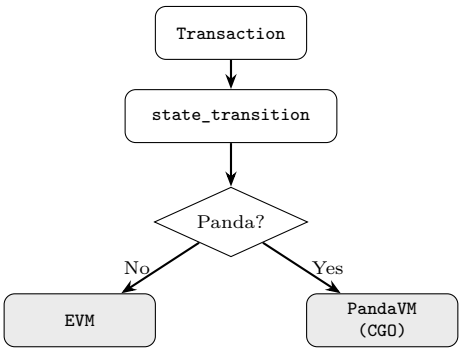

Figure 7: Dual-VM transaction routing. The state transition function inspects a contract type flag on the target account. Solidity contracts follow the standard EVM path; Python contracts are routed to PandaVM via CGO.

Table 8: Panda loader instruction set

Disc.	Instruction	Description
0	Deploy	Upload, validate, constructor
1	Call	State-mutating execution
2	Query	Read-only execution
3	Upgrade	Replace code (authority only)

does not perform on-chain PDA derivation. A future version may derive PDAs deterministically for state and event accounts.

11.5 Thread-Local VM Instances

Solana validators process transactions across multiple threads. To avoid contention on the CPython Global Interpreter Lock (GIL), each validator thread maintains its own PandaVM instance via thread-local storage. This design ensures that PandaVM scales linearly with the validator's thread count without introducing GIL contention.

11.6 Compute Unit Conversion

The Solana integration uses fixed gas limits per operation type. The target conversion ratio is:

$$\text{Solana CU} = \text{PCU} \times 5 \quad \text{(configurable)} \tag{6}$$

Zero-Knowledge Proof System

12.1 Execution Traces

Every contract execution can produce a detailed execution trace capturing the full computational history. The trace includes the contract code hash, input state root, output state root, call data hash, output hash, individual execution steps (opcode, operands, stack hash, memory hash), total step count, and gas consumed. This trace serves as the witness for proof generation.

12.2 Proof Backends

PandaVM defines three zero-knowledge proof backends, each targeting different workload characteristics. The current implementation provides the backend abstraction layer and simulation mode; production integration will connect to the respective ZK proving systems:

The choice of backend is specified by the contract developer via the `@proof`

Table 9: ZK proof backend comparison

Backend	Technology	Best For
RISC Zero	RISC-V zkVM	General Python
Halo2	IPA-based SNARK	Matrix operations
SP1	Succinct's zkVM	General-purpose

decorator or selected automatically based on workload heuristics.

12.3 Validity Proofs

In validity proof mode, the execution follows this sequence:

1. Contract executes on a node, producing an execution trace.
2. The trace is submitted to the prover network (or generated locally).
3. A ZK proof π is generated attesting that the trace corresponds to a valid execution of the contract with the given inputs.
4. The proof is submitted on-chain.
5. The verifier contract checks π against the claimed state transition.

The soundness guarantee is that no computationally bounded adversary can produce a valid proof π for an incorrect state transition:

$$\Pr[\texttt{Verify}(\pi, s, s') = 1 \mid s' \neq \texttt{exec}(C, s, d)] \leq \text{negl}(\lambda)$$

where λ is the security parameter.

12.4 Fraud Proofs

In fraud proof mode, execution is accepted optimistically. A challenge window opens (configurable, typically 7 days for L2 rollup mode). During this window, any observer can re-execute the contract and, if the result differs, submit a fraud proof identifying the divergence point. An on-chain arbiter re-executes the divergent step to determine the correct outcome. The losing party forfeits a bond.

12.5 On-Chain Verification

On Ethereum, a verifier contract provides on-chain proof verification:

```
function verifyProof(
    bytes32 stateRoot,
    bytes32 outputHash,
    uint8 backendId,
    bytes calldata proof
) external returns (bool)
```

Listing 3: Verifier contract interface

On Solana, verification is handled by the native verifier within the Panda loader program.

12.6 Prover Network

Proof generation is computationally expensive and is designed to be distributed. Validators publish execution traces. Prover nodes compete to generate valid proofs. The first valid proof earns a fee. This market-based approach enables horizontal scaling of proof generation capacity independently of validation capacity.

Privacy Layer

13.1 Private Contracts

Panda supports encrypted smart contracts. When a contract is decorated with `@private`, its source code is stored on-chain in encrypted form. Execution occurs within the VM after decryption, and the execution result (state changes, return values) is re-encrypted before storage. Only the contract deployer and authorized parties can decrypt the contract code and state.

13.2 Encryption Scheme

The encryption implementation uses AES-256-GCM authenticated encryption. Keys are derived using HKDF-SHA256 from the deployer's secret and a salt, producing 32-byte keys. Nonces are deterministic:

$$\eta = H(a \| h \| \sigma)$$

where η is the deterministic nonce, H = SHA-256, a is the contract address, h is the block height, and σ = `"panda-nonce-v1"` is a domain separator.

The 16-byte GCM authentication tag ensures tamper detection — decryption fails if any byte of ciphertext or tag has been modified. State encryption operates at per-field granularity with contract-specific keys. Zero-knowledge proofs are intended to verify the correctness of encrypted execution without revealing the plaintext code, state, or intermediate values.

13.3 Differential Privacy

On-chain differential privacy is implemented using deterministic Laplace noise generation. For a query q with sensitivity Δq and privacy budget ϵ, the noisy response is:

$$\tilde{q}(D) = q(D) + \text{Lap}\left(\frac{\Delta q}{\epsilon}\right) \tag{7}$$

The Laplace distribution is sampled via inverse CDF with SHA-256-seeded uniform random values, ensuring that noise generation is deterministic across all nodes. This enables privacy-preserving aggregate queries on on-chain data without sacrificing consensus.

13.4 Federated Learning Contracts

Four privacy-preserving federated learning patterns are implemented as contract templates:

1. **Secure Aggregation**: Participants submit masked gradient updates. Pairwise masks cancel upon summation, revealing only the aggregate gradient. No individual gradient is visible on-chain.

2. **Differential Privacy**: Gradient clipping followed by calibrated Gaussian noise addition. The noise magnitude is computed from the privacy budget ϵ and the gradient norm bound.

3. **Verified Gradients**: Pedersen commitments are used to commit to gradient updates before submission. Norm verification ensures that no single participant can corrupt the aggregate by submitting an adversarially large gradient.

4. **Full Pipeline**: Combines differential privacy, secure aggregation, and gradient verification into a single training round. This represents the strongest privacy guarantee available in the system.

Rollup Architecture

14.1 Ethereum L2 Mode

PandaVM can operate as an Ethereum L2 rollup. In this mode, the `panda-rollup` sequencer—a Rust service built on Tokio and Alloy—collects Panda transactions, batches them, executes them against the current state, and submits state roots to L1. Two modes are supported: optimistic (with fraud proofs) and ZK (with validity proofs).

14.2 L1 Contracts

The rollup is anchored by four Solidity contracts on Ethereum L1:

Bridge Contract manages deposits and withdrawals between L1 and L2. Users deposit ETH on L1 to receive a credit on L2. Withdrawals require a Merkle proof against the finalized state root. The challenge window for withdrawals is:

$$W = \frac{7 \times 24 \times 3600}{12} = 50{,}400 \text{ blocks}$$

Fraud Proof Contract (`OptimisticBatchAssertions`) implements the challenge game with the following stake parameters:

$$S_{\text{seq}} \geq 1 \text{ ETH}, \qquad B_{\text{chal}} = 0.1 \text{ ETH}$$

If fraud is proven, the sequencer's stake is slashed and distributed to the challenger. If the challenge is invalid, the challenger's bond is forfeited. Unchallenged batches auto-finalize after the challenge window expires.

Registry Contract maintains a registry of deployed Panda contracts. Each entry maps a code hash to metadata including the deployer address, deployment timestamp, version number, and verification status.

Verifier Contract provides on-chain ZK proof verification for validity proof mode. The contract defines the verification interface for all three proof backends.

14.3 Batch Lifecycle

The batch lifecycle proceeds as follows. The sequencer collects transactions and constructs a batch containing the pre-state root, post-state root, transaction data, and (optionally) a validity proof. The batch is submitted to L1 via the bridge contract. A challenge window opens. If no valid fraud proof is submitted within the window, the batch is finalized and the post-state root becomes canonical. If a fraud proof is submitted, the on-chain arbiter adjudicates.

14.4 Universal Bridge and Multi-Chain Topology

The rollup architecture extends beyond a single L1↔L2 relationship to support a *universal bridge* that routes messages between multiple Panda L2 rollups through a shared L1 hub. The topology consists of five logical chains:

1. **Settlement forks** (e.g., Polygon, Arbitrum) — external EVM chains that anchor each rollup's batch posting, escrow, and data availability.

2. **Panda L1 hub** — a dedicated Panda-Geth instance that hosts the `UniversalGateway` contract and serves as the canonical routing layer for cross-rollup messaging.

3. **Panda L2 rollups** — each rollup runs a `panda-rollup` sequencer paired with a Panda-Geth execution node, anchored to one settlement fork.

The `UniversalGateway` contract on the hub provides two core primitives. *Inbound queueing* allows hub applications or governance to enqueue messages destined for a specific rollup via `enqueueInbound(rollupId, dest, payload)`, which assigns a monotonic per-rollup nonce and emits an `L1ToL2Message` event. *Outbound execution* allows messages originating on an L2 rollup to be executed on the hub via `executeOutboundMessage(rollupId, leaf, merkleProof, target, payload)`, which verifies Merkle inclusion against the rollup's latest outbound root and enforces single-spend semantics via a consumed-leaf mapping.

On each settlement fork, a `SettlementMailbox` contract mirrors inbound messages from the hub (via `receiveFromHub`) and accepts outbound messages from L2 contracts (via `emitOutboundToHub`). The outbound leaf is computed as:

$$\ell = \kappa(\texttt{abi.encode}(r,\ t,\ \kappa(p)))$$

where $\kappa = \texttt{keccak256}$, r is the rollup ID, t is the target address, and p is the payload. This provides deterministic rehashing resistant to payload-wrapping attacks.

14.5 Bridge Relayer

A stateless bridge relayer connects the hub, settlement forks, and L2 chains through a four-stage pipeline:

1. **Scan**: Poll the hub and settlement chains for new `L1ToL2Message` and `OutboundToHub` events, tracking per-chain block cursors for crash recovery.
2. **Decode**: Extract structured message fields from ABI-encoded event data, including rollup ID, nonce, destination, and variable-length payload (capped at 64 KB).
3. **Prove**: For outbound messages (L2→hub), anchor the Merkle root on the hub. Three publication paths are supported in order of trust minimization: (a) *ZK-verified*: a 388-byte Groth16 proof is submitted to a `VerifiedMerkleRootAnchor` contract, which verifies the proof on-chain and publishes the root; (b) *Timelock*: the root is proposed via `OutboundRootTimelock` and becomes executable after a configurable block delay; (c) *Direct*: the root is set by a trusted admin role (development only).
4. **Submit**: Forward inbound messages to the settlement mailbox; execute outbound messages on the hub gateway with the appropriate Merkle proof.

The relayer is designed for high-availability deployment: it is fully idempotent (reprocessing a message that has already been consumed produces a soft error, not a crash), stateless across restarts (block cursors are the only persistent state), and supports multiple concurrent instances without coordination. The relayer detects chain resets (cursor exceeds chain head) and recovers gracefully by resetting its scan position.

14.6 Sequencer Service

The `panda-rollup` sequencer orchestrates L2 block production and L1 settlement through three concurrent async loops:

- **Block ticker**: Drains pending transactions (up to a configurable batch size, default 32), executes them against the L2 Panda-Geth node via the `panda_callContract` RPC, and commits a new L2 block with receipt hashes. The state root chain is computed as $r_{n+1} = \texttt{SHA256}(r_n \,\|\, txHashes)$ with a deterministic genesis root.
- **Inbox ticker**: Polls the settlement mailbox for `InboundFromHub` events, deduplicates by $(rollupId, nonce)$, and enqueues the decoded messages as pending L2 transactions. An L2 mirror contract (`RollupInboxMirror`) validates caller authorization, rollup ID binding, and strictly-increasing nonce ordering to prevent replay attacks.
- **Batch ticker**: Aggregates completed L2 blocks into batches, reads the current batch number from the L1 bridge contract, and submits the batch with pre-state and post-state roots. Default cadence is one batch every 8 seconds, with each batch containing up to 128 transactions across multiple L2 blocks.

This architecture achieves an L2 block time of 2 seconds (configurable), throughput of up to 16 transactions per second per rollup, and end-to-end L1→L2 message latency of approximately 5.5 seconds under default configuration. L2→L1 messages settle in approximately 2 seconds (ZK path) or after the timelock delay (optimistic path, default 7-day equivalent challenge window of 50,400 blocks at 12-second L1 block time).

14.7 Dual Finality

The settlement layer supports two finality modes. In *optimistic mode*, batches are submitted with an empty proof field and enter a challenge window during which

fraud proofs may be submitted. Batch states progress through `Submitted` → `Finalized` (if unchallenged) or `Submitted` → `Challenged` → `Rejected` (if fraud is proven). Sequencer stake (≥ 1 ETH) is slashed on proven fraud; challenger bond (0.1 ETH) is forfeited on invalid challenges.

In *ZK mode*, batches include a validity proof verified on-chain by the `PandaVerifier` contract. Valid proofs bypass the challenge window entirely, providing instant finality. The system supports Groth16 proofs (388 bytes, 500K–1M gas for on-chain verification) with the three ZK backends described in Section .

The hybrid design allows a rollup to start in optimistic mode (lower operational cost, delayed finality) and upgrade to ZK mode (higher per-batch cost, instant finality) without contract migration — both paths settle through the same `PandaBridge` contract.

Token Standards & On-Chain Primitives

15.1 PRC-20: Fungible Token Standard

PRC-20 is the Python equivalent of ERC-20, providing a standard interface for fungible tokens. The standard includes `transfer`, `approve`, `transfer_from`, `balance_of`, `total_supply`, and `allowance` methods, along with `Transfer` and `Approval` events. Implementing PRC-20 in Python allows data scientists to create and manage tokens without learning Solidity.

15.2 PRC-721: Non-Fungible Token Standard

PRC-721 is the Python equivalent of ERC-721 for non-fungible tokens. The standard includes `mint`, `transfer_from`, `owner_of`, `balance_of`, and `token_uri` methods, along with `Transfer`, `Approval`, and `ApprovalForAll` events.

15.3 Contract Patterns

The Panda contract library includes reusable patterns implemented as Python contracts:

- **Ownable**: Two-step ownership transfer with pending owner confirmation.
- **AccessControl**: Role-based permissions with admin hierarchy.
- **Pausable**: Emergency pause mechanism for contract operations.
- **Registry**: On-chain contract discovery and metadata storage.
- **DAO Voting**: Token-weighted governance with configurable quorum thresholds.
- **Escrow**: Three-party payment settlement with release and refund.
- **Multisig**: M-of-N signature schemes for multi-party authorization.

15.4 Cryptographic Primitives

The `panda.crypto` standard library provides on-chain cryptographic operations: SHA-256, SHA3-256, Keccak-256 (EVM-compatible), BLAKE2b, RIPEMD-160, double SHA-256, HMAC-SHA256 with constant-time verification, Merkle tree construction and proof verification, and Pedersen commitments for commit-reveal schemes.

15.5 PRC-Agent: AI Agent Standard (Planned)

PRC-Agent is a planned standard interface for on-chain autonomous agents. The specification calls for task submission with payment, verifiable model inference, model weight updates, and composable agent-to-agent interactions. Agents would be contracts that hold trained models in state and expose inference endpoints as `@query` methods with optional `@proof` verification. This standard is not yet implemented.

Machine Learning Contracts

16.1 panda.ml Standard Library

The `panda.ml` module provides a pure-Python ML library with scikit-learn backend fallback. When scikit-learn is available in the environment, it is used as the computation backend. When it is not (e.g., in constrained execution environments), a pure-Python fallback implementation is used, ensuring consistent API behavior.

The model lifecycle follows a standard pattern. Models are instantiated from `panda.ml` classes, trained via `fit()`, serialized via `save_model()` to a dictionary, stored in contract state, restored via `load_model()`, and used for inference via `predict()`.

All `panda.ml` floating-point results (model coefficients, metrics, predictions) are rounded to 10 decimal places to ensure deterministic behavior across nodes with different floating-point implementations.

16.2 Example: On-Chain Fraud Detection

```
@contract
class FraudDetector:
    class State:
        model: dict = {}
        training_count: int = 0

    @call
    def train(self, ctx,
              transactions: list,
              labels: list):
        from panda.ml import (
            RandomForest, save_model
        )
        model = RandomForest(
            n_estimators=10, max_depth=5
        )
        model.fit(transactions, labels)
        self.state.model = save_model(model)
        self.state.training_count += 1

    @query
    @proof(type="validity")
    def classify(self,
                 transaction: list) -> dict:
        from panda.ml import load_model
        model = load_model(self.state.model)
        pred = model.predict([transaction])
        return {
            "is_fraud": bool(pred[0]),
            "confidence": 0.95
        }
```

Listing 4: Fraud detection contract

This contract trains a random forest classifier on-chain and exposes a ZK-verifiable inference endpoint. The training data, model weights, and inference results are all on-chain and auditable.

16.3 Example: On-Chain Price Prediction

```
@contract
class PricePredictor:
    class State:
        asset_name: str = ""
        model: dict = {}
        is_trained: bool = False
        sample_count: int = 0
        last_r2: float = 0.0

    @call
    def train(self, ctx,
              features: list,
              prices: list):
        from panda.ml import (
            LinearRegression, save_model,
            r2_score
        )
        model = LinearRegression()
        model.fit(features, prices)
        preds = model.predict(features)
        self.state.model = save_model(model)
        self.state.is_trained = True
        self.state.last_r2 = round(
            r2_score(prices, preds), 6
        )

    @query
    def predict(self,
                features: list) -> list:
        from panda.ml import load_model
        model = load_model(self.state.model)
        return [round(p, 2)
                for p in model.predict(features)]
```

Listing 5: Price prediction contract with linear regression

The price predictor demonstrates the `panda.ml` lifecycle for regression models: train on-chain, persist via `save_model()`, and serve predictions through a `@query` endpoint. The R^2 score is recorded on-chain for auditability.

Table 10: Supported ML algorithms

Category	Algorithms
Regression	Linear, Polynomial
Classification	Logistic, DT, RF, KNN
Clustering	K-Means
Neural Nets	MLP, PyTorch, Keras
Ensemble	Random Forest
Online	SGD Regressor
Preprocessing	StandardScaler, PolynomialFeatures

16.4 Example Contract Library

Beyond the ML-focused contracts above, the Panda contract library includes a comprehensive set of example contracts spanning multiple application domains:

- **Counter** (94 lines): The simplest Panda contract, demonstrating `@contract`, `@constructor`, `@call`, `@query`, and `@event` decorators with owner-only reset and event emission.
- **PredictionMarket** (181 lines): A binary prediction market with time-based betting, proportional payouts, owner-only resolution, and pool-weighted odds calculation.
- **FraudDetector** (132 lines): Logistic regression classifier for real-time transaction fraud scoring with configurable threshold and on-chain prediction statistics.
- **PricePredictor** (96 lines): Linear regression model for asset price prediction with R^2 score tracking and auditable training history.
- **CrossChainSwap**: Atomic token exchange across L2 rollups using `@receiver` and `await chain.call()`.
- **TokenBridge**: Deposit/withdrawal bridge between Ethereum L1 and Panda L2 with Merkle proof verification.

All example contracts are deployed and smoke-tested on every chain boot via the CI/CD pipeline, with 69 integration tests validating the full deploy-call-query lifecycle.

16.5 Model Marketplace

The contract library includes a model marketplace pattern. A registry contract allows model developers to register trained models with metadata (algorithm type, training dataset hash, accuracy metrics). Other contracts can query the marketplace and invoke inference on registered models. Model integrity is verified via `panda.ml.load_model()`, which checks the serialized model format before deserialization.

16.6 Distributed Weight Storage for Large Models

A fundamental limitation of the contract state model is the per-contract size cap of 10 MB. Under MessagePack serialization with `float64` encoding, each parameter consumes approximately 9 bytes, restricting a single contract to roughly 1.1 million parameters. While sufficient for classical ML models such as logistic regression or random forests, this constraint excludes the class of neural network architectures that define contemporary machine learning: GPT-2 Small requires 124 million parameters, LLaMA 8B requires 8 billion, and frontier models exceed one trillion.

We observe that neural network architectures exhibit a natural decomposition into mathematically independent com-

ponents, each of which can be stored and updated in isolation. Consider a transformer model with L layers, vocabulary size V, and hidden dimension D. The model decomposes as follows:

- **Embedding weights** $\mathbf{E} \in \mathbb{R}^{V \times D}$: Row-based lookups are independent per token. Each row $\mathbf{e}_i$ is accessed only when token i appears in the input sequence.
- **Transformer blocks** $\{B_0, B_1, \ldots, B_{L-1}\}$: Each block is a self-contained residual unit comprising layer normalization, multi-head self-attention, a second layer normalization, and a feed-forward network (MLP). The residual connection ensures that each block computes $\mathbf{x}_{l+1} = \mathbf{x}_l + B_l(\mathbf{x}_l)$, where the weights of B_l are accessed only during the forward and backward passes of layer l.
- **Language model head** $\mathbf{W}_{\text{head}} \in \mathbb{R}^{D \times V}$: A single linear projection from hidden state to vocabulary logits, used only at the output stage.

This decomposition motivates a *distributed weight storage* architecture in which each component is stored in a separate `WeightShard` contract. A `Coordinator` contract orchestrates training by gathering weights via `query_contract`, performing forward and backward passes locally, and scattering gradient updates via `call_contract`. Each shard applies stochastic gradient descent independently upon receiving its gradient tensor.

The `WeightManager` SDK abstracts the distributed storage protocol:

```
from panda.distributed import WeightManager

wm = WeightManager(shard_addresses)
W = wm.gather()          # query all shards
# ... forward/backward pass ...
wm.scatter_gradients(grads, lr=0.01) # call each
    shard
```

Listing 6: WeightManager SDK for distributed training

Correctness guarantee. We verify that the distributed architecture preserves numerical equivalence with monolithic training. Specifically, the initial loss computed by the distributed system (gathering weights from N shards, assembling the full parameter vector, and computing the forward pass) matches the loss computed by a single-contract monolithic implementation to within 10^{-10} absolute error. Both configurations are constructed with identical random seeds, and loss equivalence is asserted across the gather-compute-scatter boundary.

16.7 Pipelined Multi-Transaction Training

The distributed weight storage architecture enables a second optimization: pipelined training across multiple blockchain transactions. We observe that blockchain blocks provide a natural scheduling primitive analogous to pipeline stages in hardware design. Each transaction processes a bounded unit of computation (one layer's forward or backward pass), and the deterministic block ordering guarantees correct sequencing without explicit synchronization.

For a model with L transformer layers, a single training step decomposes into $2L + 2$ transactions:

1. **Embedding forward** (1 transaction): Token embedding lookup and positional encoding.
2. **Layer forward passes** (L transactions): Each transaction t_i for $i \in \{1, \ldots, L\}$ computes the forward pass through transformer block B_{i-1}, storing intermediate activations in the Coordinator's state.
3. **Head forward + loss** (1 transaction): Language model head projection, loss computation, and initiation of the backward pass through the head.
4. **Layer backward passes** (L transactions): Each transaction t_{L+2+j} for

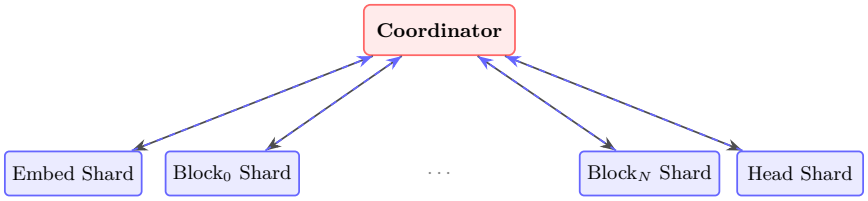

Figure 8: Distributed weight storage architecture. The Coordinator gathers weights via `query_contract` (solid arrows) and scatters gradient updates via `call_contract` (dashed arrows). Each WeightShard applies SGD independently.

$j \in \{0, \ldots, L-1\}$ computes the backward pass through block B_{L-1-j} and scatters gradient updates to the corresponding shard.

Timer-based auto-scheduling triggers each successive pipeline stage automatically. Gas consumption per transaction is bounded and predictable, as each transaction processes exactly one layer rather than the full model.

Activation checkpointing. Intermediate activations must persist in the Coordinator's contract state between transactions. For a sequence of length S tokens and hidden dimension D, the activation tensor for each layer requires $S \times D \times 9$ bytes under `float64` MessagePack encoding. For representative dimensions ($S = 128$, $D = 4096$), this yields approximately 4.7 MB per layer, well within the 10 MB contract state limit. Larger configurations may require gradient checkpointing strategies that recompute activations during the backward pass rather than storing them.

16.8 Parameter-Efficient Fine-Tuning via LoRA

For large pretrained models, full fine-tuning is prohibitively expensive in terms of both storage and computation. We adopt Low-Rank Adaptation (LoRA) [29] to reduce the number of trainable parameters by several orders of magnitude. The key insight is that weight updates during fine-tuning exhibit low intrinsic rank: the effective weight matrix can be decomposed as

$$\mathbf{W}_{\text{eff}} = \mathbf{W}_{\text{base}} + \frac{\alpha}{r}\mathbf{AB} \tag{8}$$

where $\mathbf{W}_{\text{base}} \in \mathbb{R}^{D \times D}$ is the frozen pretrained weight matrix, $\mathbf{A} \in \mathbb{R}^{D \times r}$ and $\mathbf{B} \in \mathbb{R}^{r \times D}$ are the low-rank adapter matrices, $r \ll D$ is the adapter rank, and α is a scaling hyperparameter.

In the distributed architecture, the frozen base weights $\mathbf{W}_{\text{base}}$ are stored in weight shards using quantized representations (Section 16.9) and are never updated. Only the adapter matrices $\mathbf{A}$ and $\mathbf{B}$ receive gradient updates. For a model with hidden dimension D and adapter rank r, each attention projection requires $2 \times D \times r$ trainable parameters per adapter. This yields dramatic reductions in the number of shards required for training.

Example: LLaMA 8B. A LLaMA 8B model with 32 transformer layers, each containing four attention projections (Q, K, V, O), requires LoRA adapters on 128 weight matrices. With rank $r = 16$ and hidden dimension $D = 4096$, each adapter contains $2 \times 4096 \times 16 = 131{,}072$ parameters. The total trainable parameter count is $128 \times 131{,}072 = 16.8$ million, requiring 17 shards under `int8` quantization — compared to approximately 8,000 shards for the full `float64` model.

16.9 Quantized Weight Storage

Quantization reduces the per-parameter storage cost, enabling more parameters per shard. We support four quantization levels, each offering a different trade-off between precision and storage density.

For `int8` quantization, the encod-

ing is:

$$q = \text{round}\left(\frac{w}{s}\right) + z \tag{9}$$

where w is the original weight value, s is the scale factor, z is the zero point, and q is the quantized integer. Dequantization recovers $\hat{w} = s \cdot (q - z)$.

Quantization is applied at the shard level: each `WeightShard` stores its weights in the specified format and performs dequantization on read. This is transparent to the Coordinator, which always operates on full-precision tensors during forward and backward passes.

16.10 Scaling Analysis

The combination of distributed weight storage, LoRA adapters, quantized storage, and pipelined transactions enables a progression from small models that are fully operational today to large models that are architecturally feasible. Table 12 summarizes the scaling characteristics.

The NanoGPT configuration (33K parameters, single shard) has been fully implemented and tested. Distributed training produces loss values matching the monolithic implementation to within 10^{-10}, and two identical distributed runs produce byte-identical contract states, confirming full determinism. The GPT-2 Small configuration is architecturally feasible with current infrastructure: 7 base weight shards under `int4` quantization and 2 LoRA adapter shards, with 24 transactions per training step. The LLaMA configurations represent longer-term targets that are architecturally sound but require further optimization of cross-contract call overhead.

16.11 Automatic Training Orchestration

The distributed training infrastructure described above — weight sharding, pipelined transactions, LoRA adapters — provides the low-level primitives for on-chain model training. However, orchestrating a complete training loop still requires the developer to manually decompose iterations across blocks, manage checkpointing between transactions, split datasets into per-block mini-batches, and aggregate gradients across worker contracts. This manual decomposition is error-prone and represents a significant barrier to adoption, particularly for data scientists unfamiliar with blockchain transaction semantics.

We address this with four contracts that automate the training lifecycle, building on the deterministic async/await primitives (Section 8) and the cross-contract call infrastructure (Section).

AutoCheckpointTrainer. The core pattern exploits Python's `async/await` with `await sleep(blocks=1)` to automatically distribute a training loop across blocks. Each block processes exactly one mini-batch, and model weights are checkpointed to contract state between blocks. The developer writes a standard training loop; the CPS transformation (Section 8) handles the cross-block suspension transparently. Training is resumable via `resume_training()` after node restarts or gas exhaustion, and cancellable via `cancel_training()`. Progress — including current epoch, batch index, and running loss — is queryable mid-flight through a `@query` method. The following listing illustrates the core pattern:

```
from panda import contract, call, query, sleep
from panda.ml import LinearRegression
from panda.training import DataLoader,
    CheckpointState

@contract
class AutoCheckpointTrainer:
    class State:
        model: dict = {}
        checkpoint: dict = {}
        status: str = "idle"

    @call
    async def train(self, ctx, data, labels,
                    epochs=10, batch_size=32):
        loader = DataLoader(data, labels,
                            batch_size=batch_size)
        ckpt = CheckpointState(self.state)
        model = LinearRegression()

        for epoch in range(ckpt.epoch, epochs):
            for X_b, y_b in
                loader.batches(ckpt.batch):
                model.fit(X_b, y_b)
                ckpt.save(model, epoch, loader.pos)
                await sleep(blocks=1) # yield block
```

Table 11: Storage density by quantization format

Format	Bytes/param	Params per 10 MB shard
`float64` (msgpack)	9	1.1M
`float32`	4	2.5M
`int8` + scale/zero	2	5.0M
`int4` (packed)	0.5	20.0M

Table 12: Scaling analysis: distributed training feasibility across model sizes

Model	Params	Base Shards (int4)	LoRA Shards	Txs/Step	Status
NanoGPT	33K	1	—	1	Working (tested)
GPT-2 Small	124M	7	2	24	Feasible
LLaMA 8B	8B	400	17	64	Ambitious
LLaMA 70B	70.6B	3,530	150	160	Theoretical

```
        ckpt.reset_batch()

    self.state.model = model.to_dict()
    self.state.status = "complete"

@call
async def resume_training(self, ctx):
    ckpt = CheckpointState(self.state)
    # resumes from last checkpoint
        automatically
    await self._continue(ctx, ckpt)

@query
def progress(self, ctx):
    ckpt = CheckpointState(self.state)
    return {"epoch": ckpt.epoch,
            "batch": ckpt.batch,
            "status": self.state.status}
```

Listing 7: AutoCheckpointTrainer: automatic cross-block training

DataParallelTrainer and TrainingWorker. For compute-bound workloads, we implement the standard data-parallel pattern using cross-contract calls. A `DataParallelTrainer` coordinator contract maintains a registry of `TrainingWorker` contracts. On each training step, the coordinator partitions the current mini-batch into N shards (one per registered worker), dispatches each shard via `call_contract`, and collects the resulting gradient tensors via `query_contract`. The coordinator computes the all-reduce mean of the gradients:

$$\bar{\mathbf{g}} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{g}_i \tag{10}$$

and applies a single SGD update $\mathbf{w} \leftarrow \mathbf{w} - \eta\bar{\mathbf{g}}$. When no workers are registered ($N = 0$), the coordinator falls back to single-contract training, computing gradients locally. This design mirrors the parameter server architecture common in distributed ML systems but operates entirely within the blockchain's deterministic execution model.

AutoShardTrainer. This contract combines the distributed weight storage architecture (Section) with automatic gradient scatter/gather, making the distribution transparent to the caller. A 2-layer neural network's weights are stored across `WeightShard` contracts (as described in the distributed weight storage subsection). When the user calls `train(x, y)`, the coordinator gathers all weight matrices from shards via `query_contract`, assembles the full parameter vector, performs the forward pass and gradient computation locally, then scatters gradient updates back to the respective shards via `call_contract`. Each shard applies its SGD update independently. The user interacts with a single contract address and a simple `train`/`predict` interface; the multi-contract orchestration is entirely hidden.

SDK utilities. The `panda.training` module provides four utilities that support the orchestration contracts:

- `DataLoader`: Sliding-window batching over datasets with epoch boundary management. Tracks the current position within the dataset and supports resumption from an arbitrary offset, enabling seamless integration with checkpointing.
- `CheckpointState`: Resumable progress tracking that serializes and restores training state (current epoch, batch index, model weights, running loss) to and from contract state. Provides `save()` and `restore()` methods that write to the contract's `State` object.
- `estimate_model_size(model)`: Pre-deployment state size checker that computes the serialized size of a model's parameters and compares against the 10 MB contract state limit. Returns the estimated size in bytes and a boolean indicating whether deployment is feasible within a single contract or requires weight sharding.
- `partition_epochs(total_epochs, batch_count)`: Block planning utility that computes the total number of blocks required for a multi-block training run, accounting for one `await sleep(blocks=1)` per mini-batch. Returns a schedule mapping block offsets to (epoch, batch) pairs, useful for gas estimation and progress monitoring.

Test coverage. The automatic training orchestration layer is validated by 117 tests: 89 contract harness tests covering the `AutoCheckpointTrainer`, `DataParallelTrainer`, `TrainingWorker`, and `AutoShardTrainer` contracts (deployment, training execution, checkpoint resumption, cancellation, progress queries, multi-worker gradient aggregation, single-worker fallback, and shard gather/scatter correctness), and 28 SDK unit tests covering `DataLoader` batching and epoch boundaries, `CheckpointState` serialization round-trips, `estimate_model_size` accuracy against known model configurations, and `partition_epochs` schedule generation.

Deployment & Infrastructure

17.1 Deployment Modes

Panda supports four deployment modes:

17.2 Kubernetes Architecture

For local and production deployments, Panda provides a single-command Kubernetes setup. The deployment includes Geth validators, Solana validators, the block explorer, prover nodes, monitoring, and a faucet service.

17.3 CLI Tooling

The `panda` CLI provides developer tooling for the full contract lifecycle:

```
panda deploy contract.py
panda call <addr> <method> [args]
panda query <addr> <method> [args]
panda lint contract.py
panda test contract.py
```

The `lint` command runs static analysis locally before deployment. The `test` command executes the contract in a local `PandaVM` instance with a mock execution context, enabling rapid iteration without chain interaction.

17.4 Container Security

All deployment configurations enforce defense-in-depth at the container level: read-only root filesystems, all Linux capabilities dropped, privilege escalation prevention, bounded temporary storage, non-root execution, and CPU/memory resource limits per node.

Multi-node deployments run each validator in an isolated container on a private network with static IPs, enabling cross-validator determinism testing. These container-level restrictions serve as the outermost security perimeter: even if a contract escapes PandaVM's sandbox entirely

Table 13: Deployment modes

Mode	Description
ETH L2 Rollup	Rollup with sequencer and bridge
ETH Standalone	Direct Geth integration
Solana Native	Builtin in Agave validator
Local Dev	Docker Compose or Kind

(bypassing all eleven internal layers), the container itself has no capabilities, no writable filesystem except bounded tmpfs, no privilege escalation path, and restricted network access.

Evaluation

The system is evaluated across six dimensions: security, determinism, integration, async execution, stress testing, and fuzzing.

18.1 Security Evaluation

The security model was evaluated through adversarial testing across 12 attack categories. Attack categories are summarized in Table 14.

Gas metering bypass is validated empirically. The bounded exponentiation cap blocks large exponents while permitting normal arithmetic and modular exponentiation. Memory bomb defense catches both list and string repetition attacks through at least one defense layer. Recursion depth limits reject unbounded recursion while permitting normal call depths.

Risk mitigation is validated empirically. Gas determinism is verified by executing contracts with standard library imports multiple consecutive times and asserting identical gas consumption on every run. Cross-contract import isolation is verified by interleaving executions of two contracts and asserting that interleaved gas matches standalone gas. NaN and Infinity injection are verified to produce clean errors rather than crash or silent corruption.

18.2 Determinism Verification

Determinism is validated by executing each scenario multiple consecutive times with identical inputs and asserting byte-for-byte identity of state, output, and events across all runs. Scenarios cover: basic arithmetic, floating-point operations, dictionary ordering under hash randomization, set operations, list comprehensions, string formatting, sort stability, math library functions, seeded random number generation, large integer arithmetic, nested state mutations, and ML model training.

Floating-point determinism is further validated for NumPy, pandas, and timer scheduling workloads. Cross-platform reproducibility is verified across Linux and macOS environments.

18.3 Async/Await Verification

The deterministic async/await system is validated against each security claim. Table 15 summarizes the properties verified.

18.4 Stress Testing and Fuzzing

Sustained throughput is validated through stress testing, including rapid-fire execution tests that verify no performance degradation under sustained load. Fuzzing coverage exercises edge cases in contract

Table 14: Security test evidence by attack category

Attack Category	Example Scenarios
Sandbox escape (direct)	Forbidden imports, eval, reflection chains
Sandbox escape (indirect)	Concurrency, signal, FFI, HTTP modules
Network / IPC	RPC, SMTP, FTP, Telnet
Non-determinism	Time overrides, random seeding, UUID
Gas / resource exhaustion	Infinite loops, memory bombs, recursion
State corruption	Query mutation, type confusion, oversized state
Injection	JSON injection, method name injection, format strings
Token security	Integer overflow, double-spend, negative transfer
Crypto security	Hash collision, timing-safe HMAC, Merkle forgery
Storage security	Type enforcement, counter overflow, queue boundary
Cross-contract	Reentrancy, delegatecall prevention
Edge cases	Null bytes, Unicode, deeply nested state

Table 15: Async/await security claims with test evidence

Claim	Assertion
Replay attacks prevented	Replayed resume starts from checkpoint 0
Caller identity preserved	Caller matches across timer boundaries
Code mutation detected	SHA-256 mismatch raises hard error
Reentrancy blocked	Second call rejected while hibernated
Phase isolation enforced	Post-await code unreachable in Phase 0
Timer determinism	Identical timer IDs across runs
Gas determinism per phase	Byte-for-byte gas match across runs
Nested composition	Return values propagate across levels
Circular dependency detection	Static analysis rejects at load time
Forbidden await targets	Static AST rejection

parsing, state serialization, gas metering, and execution isolation with randomized inputs.

18.5 Distributed Training Evaluation

The distributed weight storage and training pipeline is validated across three properties: monolithic equivalence, determinism, and structural correctness.

Monolithic equivalence. A NanoGPT model (33K parameters) is constructed in both monolithic (single-contract) and distributed (multi-shard) configurations with identical random seeds. The initial forward pass loss values match to within 10^{-10} absolute error, confirming that the gather-compute-scatter protocol introduces no numerical divergence.

Determinism. Two identical distributed training runs are executed with the same inputs and random seeds. Byte-for-byte identity of all shard states, coordinator state, computed losses, and gradient tensors is verified across both runs. This validates that the distributed training pipeline preserves the determinism guarantees of the underlying PandaVM.

Structural correctness. All distributed training contracts — weight shards, coordinator, LoRA adapters, and quantized storage — are validated for correct decorator usage, state field definitions, method signatures, and full deploy-call-query lifecycle behavior, including cross-contract weight gathering and gradient scattering.

18.6 Known Limitations

In-process memory limits. Address space limits cannot be applied in the default in-process execution mode because they restrict the entire host process, which would prevent ML library loading. Memory bomb defense in-process relies on bounded exponentiation, recursion limits, and wall-clock timeouts. Full address space enforcement is available in forked sandbox mode.

OS-level sandbox requires opt-in. The seccomp-BPF filter (L7) is available but defaults to off. Full OS-level isolation — capability dropping (L8), namespace isolation (L9), address space limits, and file descriptor cleanup — requires the forked sandbox mode, which is feature-gated and not enabled by default. In the default in-process mode without seccomp, C-extension I/O defense relies solely on Python-level module patching (L6).

Platform-specific constraints. Seccomp-BPF is Linux-specific. On macOS, a platform sandbox profile is defined for the forked path but the in-process seccomp path is a no-op. On Windows, no kernel-level sandbox exists. Non-Linux platforms fall back to Python-level sandboxing only.

Future Work

19.1 Achieved Since Initial Design

Several items previously identified as future work have been implemented and validated. Distributed weight storage across multiple contracts enables neural network models exceeding the 10 MB per-contract state limit. Parameter-efficient fine-tuning via LoRA reduces trainable parameters by orders of magnitude for large pretrained models. Quantized weight storage (`int8` and `int4`) increases storage density by up to 18× relative to `float64` encoding. Pipelined multi-transaction training decomposes training steps across blockchain transactions with bounded per-transaction gas cost. These contributions are detailed in Section . A universal bridge relayer enables bidirectional cross-chain messaging between multiple Panda L2 rollups through a shared L1 hub, with dual finality paths (optimistic with 7-day challenge window, ZK with instant finality via Groth16 proofs). The four-stage scan-decode-prove-submit pipeline is stateless, idempotent, and supports high-availability deployment with multiple concurrent relayer instances.

19.2 Near-Term

Immediate priorities include: enabling seccomp and forked sandbox mode by default in production deployments; a custom CPython build with compile-time removal of dangerous modules (strictly superior to runtime restriction); seccomp argument filtering to restrict file operations to read-only during pre-warming; in-process address space enforcement via a memory reservation strategy that avoids interfering with ML library loading; cross-platform sandbox support; and scaling distributed training to GPT-2 Small (124M parameters, 7 base shards).

19.3 Medium-Term

Over the medium term, the project aims to integrate a fully homomorphic encryption (FHE) bridge for private computation over encrypted model weights, enable cross-chain contract migration with state portability between Ethereum and Solana, build a decentralized model registry with reputation scoring and community curation, develop on-chain AutoML pipelines that automate model selection and hyperparameter optimization, and implement tensor parallelism within individual weight matrices to support models whose single-layer parameters exceed the 10 MB shard limit.

19.4 Long-Term

Long-term research directions include appchain mode for dedicated chains optimized for specific data science work-

loads, multi-VM support extending beyond Python to R and Julia, hardware acceleration with deterministic GPU execution guarantees, development of a standardized verifiable data science protocol that other blockchain ecosystems can adopt, Mixture-of-Experts (MoE)-aware sparse activation gathering that skips inactive expert shards during distributed inference, on-chain knowledge distillation pipelines that train compact student models from large teacher models stored across weight shards, and hardware-accelerated dequantization for `int4` weights at the VM level.

Conclusion

This paper has presented Panda, a verifiable data science layer for Ethereum and Solana. The system makes the following contributions. PandaVM solves the determinism problem for scientific computing on blockchain through a six-layer enforcement architecture. An eleven-layer security model provides defense-in-depth for untrusted Python code execution across two graduated execution modes: in-process (seven layers always active, with optional kernel-level syscall filtering) and forked sandbox (all eleven layers active, including memory bounds, capability dropping, and namespace isolation). A novel two-phase seccomp-BPF filter architecture provides kernel-level syscall confinement while accommodating ML library initialization requirements. Deterministic async/await — the first cross-block execution primitive on any blockchain — enables long-running computations through CPS and checkpoint-resume modes with a dedicated ten-layer security model addressing replay, spoofing, code mutation, and reentrancy. Dual-chain portability is achieved through a standalone Rust library with a C ABI, integrated into Geth via CGO and into Agave natively. Three ZK proof backends (RISC Zero, Halo2, SP1) provide a framework for verifiable ML training and inference with succinct proofs. Privacy-preserving primitives, including federated learning, differential privacy, and encrypted contracts, are first-class citizens of the protocol. The Python SDK provides a familiar development experience for data scientists through decorators and a standard library that includes NumPy, pandas, scikit-learn, TensorFlow, PyTorch, Keras, SciPy, and XGBoost. Distributed weight storage across multiple contracts, combined with LoRA parameter-efficient fine-tuning and quantized representations, extends the system's reach from classical ML models to neural networks with billions of parameters — a novel contribution that demonstrates the viability of on-chain deep learning within the constraints of blockchain state management. A universal bridge relayer with a four-stage pipeline enables bidirectional cross-chain messaging between multiple L2 rollups through a shared L1 hub, with dual finality paths supporting both optimistic settlement and instant ZK-verified finality.

Panda positions itself as the verifiable data science layer for the multi-chain future. By enabling data scientists to deploy Python smart contracts with cryptographic guarantees of correctness, the system bridges the gap between the scientific computing ecosystem and trustless blockchain infrastructure. The result is a platform where models can be trained, evaluated, and served on-chain with the same auditability guarantees that blockchain provides for financial transactions.

Bibliography

[1] Nakamoto, S.
Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[2] Wood, G.
Ethereum: A Secure Decentralised Generalised Transaction Ledger (Yellow Paper), 2014.

[3] Yakovenko, A.
Solana: A New Architecture for a High Performance Blockchain, 2018.

[4] McMahan, H.B., Moore, E., Ramage, D., Hampson, S., Arcas, B.A.
Communication-Efficient Learning of Deep Networks from Decentralized Data. AISTATS, 2017.

[5] Dwork, C.
Differential Privacy. ICALP, 2006.

[6] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.
Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. USENIX Security, 2014.

[7] Bowe, S., Grigg, J., Hopwood, D.
Halo: Recursive Proof Composition without a Trusted Setup, 2019.

[8] Boneh, D., Drake, J., Fisch, B., Gabizon, A.
Recursive Proof Composition from Accumulation Schemes. TCC, 2020.

[9] IEEE Computer Society.
IEEE Standard for Floating-Point Arithmetic (IEEE 754-2019), 2019.

[10] Python Software Foundation.
CPython 3.12 Reference Implementation, 2023.

[11] Gensyn.
Gensyn: A Protocol for Training Machine Learning Models, 2023.

[12] Bittensor Foundation.
Bittensor: A Peer-to-Peer Intelligence Market, 2023.

[13] Ocean Protocol Foundation.
Ocean Protocol: A Decentralized Data Exchange Protocol, 2020.

[14] SingularityNET.
SingularityNET: A Decentralized, Open Market and Network for AI Services, 2019.

[15] RISC Zero.
RISC Zero: General-Purpose Zero-Knowledge Virtual Machine, 2023.

[16] Succinct Labs.
SP1: A Performance-Focused zkVM, 2024.

[17] European Parliament and Council.
Regulation (EU) 2024/1689: Artificial Intelligence Act, 2024.

[18] Corbet, J.
A seccomp overview. LWN.net, 2015.

[19] Ming, J., Verner, E., Sarwate, A., Kelly, R., Reed, C., Kahleck, T., Silva, R., Panta, S., Turner, J., Plis, S., Calhoun, V.
COINSTAC: Decentralizing the future of brain imaging analysis. F1000Research, 6:1512, 2017.

[20] Chromium Project.
Linux Sandboxing: Seccomp-BPF Integration. Chromium Design Documents, 2012.

[21] Young, E., Ghemawat, S., Burrows, M.
gVisor: Container Runtime Sandbox. Google, 2018.

[22] Near Protocol.
Near Protocol: Scalable Decentralized Applications Platform, 2020.

[23] Confio.
CosmWasm: Smart Contracts for the Cosmos Ecosystem, 2021.

[24] Cartesi Foundation.
Cartesi: The Blockchain OS, 2022.

[25] Aztec Protocol.
Aztec: Encrypted Ethereum, 2023.

[26] Optimism Foundation.
The Optimistic Rollup, 2021.

[27] Offchain Labs.
Arbitrum: Scalable, Private Smart Contracts. USENIX Security, 2018.

[28] PyO3 Contributors.
PyO3: Rust Bindings for the Python Interpreter, 2023.

[29] Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.
LoRA: Low-Rank Adaptation of Large Language Models. ICLR, 2022.

[30] Dettmers, T., Lewis, M., Belkada, Y., Zettlemoyer, L.
LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. NeurIPS, 2022.

Appendix B: Feature Parity Matrix (EVM vs. Solana)

Table 16: Feature parity across chain integrations

Feature	ETH	SOL	Parity
Deploy	Yes	Yes	Full
Call (mutating)	Yes	Yes	Full
Query (read-only)	Yes	Yes	Full
Upgrades	Yes	Yes	Full
Gas (PCU)	×10	Hardcoded	Adapted
Cross-contract	Yes	Planned	Partial
Events	EVM logs	Prog. logs	Full
ZK verification	Solidity	Planned	Partial
Fraud proofs	Yes	N/A	ETH only
L2 rollup	Yes	N/A	ETH only
Bridge	Yes	N/A	ETH only
State storage	Trie	PDA	Adapted
VM integration	CGO	Native	Adapted
Activation	Time-based	Builtin	Adapted

Appendix C: Forbidden Module List

Panda contracts are prohibited from importing 64 Python modules. Each is blocked at both the static analysis layer (L1) and the runtime import hook (L4). The runtime hook additionally catches submodule imports (e.g., `os.path`) and dynamic imports via string construction. The forbidden modules are organized by threat category:

Table 17: Forbidden Python module categories

Category	Count	Rationale
System access (`os`, `sys`, `signal`, etc.)	14	Host process and OS interaction
Filesystem (`shutil`, `pathlib`, `glob`, etc.)	4	File read/write/traverse
Network (`socket`, `http`, `urllib`, etc.)	11	TCP/UDP, HTTP, FTP, SMTP, RPC
Concurrency (`threading`, `asyncio`, etc.)	5	Non-deterministic scheduling
FFI (`ctypes`, `cffi`)	2	Bypass all Python-level restrictions
Reflection (`inspect`, `marshal`, `dis`, etc.)	6	Bytecode inspection and code generation
Debugging (`pdb`, `profile`, `trace`, etc.)	4	Execution introspection
Build tools (`setuptools`, `pip`, `venv`, etc.)	6	Package installation and build
Non-determinism (`secrets`, `uuid`)	2	Cryptographic or hardware randomness
Interactive / GUI / Misc	10	REPL, GUI frameworks, Easter eggs

Appendix D: Contract Examples

3.1 PRC-20 Fungible Token

```
from panda import contract, constructor, call, query, event

@contract
```

```python
class PRC20Token:
    class State:
        name: str = ""
        symbol: str = ""
        total_supply: int = 0
        balances: dict = {}
        allowances: dict = {}

    @constructor
    def init(self, ctx, name, symbol, initial_supply):
        self.state.name = name
        self.state.symbol = symbol
        self.state.total_supply = initial_supply
        self.state.balances = {ctx.sender: initial_supply}

    @call
    def transfer(self, ctx, to, amount):
        sender = ctx.sender
        assert self.state.balances.get(sender, 0) >= amount
        self.state.balances[sender] -= amount
        self.state.balances[to] = self.state.balances.get(to, 0) + amount
        self.emit(event.Transfer(from_addr=sender, to_addr=to, amount=amount))

    @call
    def approve(self, ctx, spender, amount):
        key = f"{ctx.sender}:{spender}"
        self.state.allowances[key] = amount
        self.emit(event.Approval(owner=ctx.sender, spender=spender, amount=amount))

    @call
    def transfer_from(self, ctx, from_addr, to, amount):
        key = f"{from_addr}:{ctx.sender}"
        assert self.state.allowances.get(key, 0) >= amount
        assert self.state.balances.get(from_addr, 0) >= amount
        self.state.allowances[key] -= amount
        self.state.balances[from_addr] -= amount
        self.state.balances[to] = self.state.balances.get(to, 0) + amount
        self.emit(event.Transfer(from_addr=from_addr, to_addr=to, amount=amount))

    @query
    def balance_of(self, owner) -> int:
        return self.state.balances.get(owner, 0)

    @query
    def total_supply(self) -> int:
        return self.state.total_supply
```

Listing 8: PRC-20 token implementation

3.2 Federated Learning Trainer

```python
from panda import contract, constructor, call, query, event

@contract
class FederatedTrainer:
    class State:
        global_model: dict = {}
        round_number: int = 0
        min_participants: int = 3
        updates: list = []
        participants: list = []

    @constructor
    def init(self, ctx, min_participants):
        self.state.min_participants = min_participants

    @call
    def submit_update(self, ctx, model_update: dict):
        assert ctx.sender not in self.state.participants
        self.state.updates.append(model_update)
        self.state.participants.append(ctx.sender)
        self.emit(event.UpdateSubmitted(
            participant=ctx.sender,
            round=self.state.round_number
        ))
```

```
    @call
    def aggregate(self, ctx):
        assert len(self.state.updates) >= self.state.min_participants
        # FedAvg: average all submitted model updates
        aggregated = {}
        n = len(self.state.updates)
        for update in self.state.updates:
            for key, value in update.items():
                if key not in aggregated:
                    aggregated[key] = 0.0
                aggregated[key] += value / n
        self.state.global_model = aggregated
        self.state.round_number += 1
        self.state.updates = []
        self.state.participants = []
        self.emit(event.RoundCompleted(round=self.state.round_number))

    @query
    def get_global_model(self) -> dict:
        return self.state.global_model
```

Listing 9: Federated learning contract

3.3 DAO Governance

```
from panda import contract, constructor, call, query, event
from panda import query_contract

@contract
class DAOVoting:
    class State:
        token_address: str = ""
        proposals: dict = {}
        proposal_count: int = 0
        quorum: int = 100
        voting_period: int = 86400

    @constructor
    def init(self, ctx, token_address, quorum):
        self.state.token_address = token_address
        self.state.quorum = quorum

    @call
    def create_proposal(self, ctx, description):
        pid = self.state.proposal_count
        self.state.proposals[str(pid)] = {
            "description": description,
            "proposer": ctx.sender,
            "yes_votes": 0,
            "no_votes": 0,
            "voters": [],
            "created_at": ctx.block_time,
            "executed": False
        }
        self.state.proposal_count += 1
        self.emit(event.ProposalCreated(id=pid, proposer=ctx.sender))

    @call
    def vote(self, ctx, proposal_id, support):
        prop = self.state.proposals[str(proposal_id)]
        assert ctx.sender not in prop["voters"]
        weight = query_contract(
            self.state.token_address,
            "balance_of", owner=ctx.sender
        )
        if support:
            prop["yes_votes"] += weight
        else:
            prop["no_votes"] += weight
        prop["voters"].append(ctx.sender)
        self.emit(event.VoteCast(
            proposal_id=proposal_id,
            voter=ctx.sender, weight=weight
        ))
```

```
    @query
    def get_proposal(self, proposal_id) -> dict:
        return self.state.proposals.get(str(proposal_id), {})
```

Listing 10: DAO voting contract

Appendix E: JSON-RPC API Reference

Table 18: Panda JSON-RPC methods (Ethereum)

Method	Parameters	Description
`panda_deploy`	source, ctor_args	Deploy a Python contract
`panda_call`	addr, method, args	Invoke state-mutating method
`panda_query`	addr, method, args	Invoke read-only method
`panda_getState`	addr	Return full contract state
`panda_getCode`	addr	Return contract source code
`panda_lint`	source	Run static analysis
`panda_estimateGas`	addr, method, args	Estimate PCU cost
`panda_getProof`	receipt_hash, backend	Get ZK proof for execution
`panda_verifyProof`	proof, state_root, output	Verify a ZK proof