NanoPy

A Python-Native Ethereum-Compatible Blockchain
with Layer 2 Scaling

Whitepaper v1.0

December 2025

Abstract

NanoPy is an Ethereum-compatible blockchain implementation written entirely in Python. It provides a fully functional Layer 1 blockchain with Proof of Stake consensus and a high-performance Layer 2 (NanoPy Turbo) for scalability. Unlike existing blockchains written in Go, Rust, or C++, NanoPy is designed for accessibility, education, and rapid prototyping by the world's largest developer community: Python developers.

```
pip install nanopy-chain
```

Contents

1 Introduction

1.1 The Problem

Existing blockchain implementations present significant barriers to entry:

- **Language Complexity**: Most blockchains are written in Go (Ethereum), Rust (Solana), or C++ (Bitcoin)—languages with steep learning curves
- **Inaccessible Codebases**: Understanding blockchain internals requires expertise in low-level systems programming
- **Limited Prototyping**: Researchers and developers cannot easily experiment with consensus mechanisms or EVM modifications

1.2 The Solution

NanoPy addresses these challenges by providing:

- **Pure Python Implementation**: Readable, modifiable blockchain code accessible to millions of Python developers
- **Full Ethereum Compatibility**: Complete EVM execution, JSON-RPC API, and MetaMask support
- **Layer 2 Scaling**: NanoPy Turbo for high-throughput applications
- **Developer Accessibility**: Simple installation via pip, run, and modify

2 Architecture

2.1 System Overview

NanoPy consists of two layers: a Python-based Layer 1 for security and decentralization, and a high-performance Layer 2 for scalability.

NanoPy L1 (Python)		
Consensus (PoS)	EVM Engine	State Storage
P2P Network	JSON-RPC API	Transaction Pool

Figure 1: Layer 1 Architecture

2.2 Layer 1: NanoPy Core

The Layer 1 is implemented entirely in Python with the following modules:

Module	Description
`nanopy.core`	Block, Transaction, Account, WorldState
`nanopy.vm`	EVM implementation with all opcodes
`nanopy.consensus`	Proof of Stake engine
`nanopy.network`	P2P networking and RPC server
`nanopy.storage`	LevelDB-backed state persistence

Table 1: Core Modules

2.3 Layer 2: NanoPy Turbo

NanoPy Turbo is an optimistic rollup implemented in Go for maximum performance:

- Executes transactions off-chain on L2
- Submits state roots to L1 every 10 blocks
- Enables withdrawals via Merkle proofs
- Provides 10x+ throughput improvement over L1

3 Consensus Mechanism

3.1 Proof of Stake

NanoPy uses a stake-weighted validator selection mechanism. The consensus parameters are defined as:

```
class ProofOfStake:
    MIN_STAKE = 10_000 * 10**18   # 10,000 NPY
    BLOCK_REWARD = 2 * 10**18     # 2 NPY per block
    LOCK_PERIOD = 100             # Blocks before unstake
```

3.2 Validator Selection

The validator selection process follows these steps:

1. Collect all active validators weighted by stake amount
2. Generate deterministic random seed using previous block hash
3. Select validator proportionally to stake weight
4. Selected validator produces the next block
5. Block rewards compound to validator stake

The selection algorithm ensures fairness while remaining deterministic:

$$P(v_i) = \frac{stake(v_i)}{\sum_{j=1}^{n} stake(v_j)} \tag{1}$$

where $P(v_i)$ is the probability of validator v_i being selected.

3.3 Slashing

Misbehaving validators are penalized through slashing:

- **Penalty**: 10% of staked amount
- **Deactivation**: Automatic if stake falls below minimum
- **Triggers**: Double signing, unavailability, invalid blocks

3.4 Bootstrap Mode

Before staking contract deployment, a bootstrap validator (founder address) can produce blocks to initialize the network. This enables network launch without requiring pre-existing stake.

4 Ethereum Virtual Machine

4.1 Full EVM Compatibility

NanoPy implements the complete Ethereum Virtual Machine specification:

- **Stack Machine**: 1024 element stack with 256-bit words
- **Memory**: Byte-addressable, dynamically expandable
- **Storage**: Key-value storage per contract (32-byte slots)
- **Opcodes**: All Shanghai-compatible opcodes

4.2 Supported Features

Feature	Status
Smart Contracts	Supported
CREATE / CREATE2	Supported
CALL / DELEGATECALL / STATICCALL	Supported
Precompiles (1-9)	Supported
EIP-1559 Transactions (Type 2)	Supported
EIP-2930 Access Lists (Type 1)	Supported
Events / Logs (LOG0-LOG4)	Supported

Table 2: EVM Feature Support

4.3 Gas Model

NanoPy follows the Ethereum gas model with standard costs:

Operation	Gas Cost
G_ZERO	0
G_BASE	2
G_VERYLOW	3
G_LOW	5
G_MID	8
G_HIGH	10
G_SLOAD	800
G_SSTORE_SET	20,000
G_SSTORE_RESET	5,000

Table 3: Gas Costs

5 Network Architecture

5.1 P2P Layer

NanoPy uses libp2p for peer-to-peer communication:

- **Protocol**: `/nanopy/1.0.0`

- **Discovery**: Bootstrap nodes + DHT
- **Propagation**: Gossip protocol for blocks and transactions

5.2 JSON-RPC API

Full Ethereum JSON-RPC compatibility enables seamless integration with existing tools:

```
eth_chainId, eth_blockNumber, eth_getBalance
eth_sendRawTransaction, eth_call, eth_estimateGas
eth_getTransactionReceipt, eth_getLogs
net_version, web3_clientVersion
```

MetaMask and other Ethereum wallets can connect directly using the RPC URL and Chain ID.

6 Layer 2: NanoPy Turbo

6.1 Sequencer

The L2 sequencer is responsible for transaction ordering and block production:

```
type Sequencer struct {
    bc         *Blockchain
    coinbase   common.Address
    blockTime  time.Duration  // 10 seconds
    l1RPC      string         // L1 connection
    l1Bridge   common.Address // Bridge contract
}
```

The sequencer:

1. Collects transactions from users
2. Executes them in deterministic order
3. Produces blocks every 10 seconds
4. Submits state roots to L1 bridge contract

6.2 Bridge Architecture

The bridge system consists of two contracts:

L1 Bridge (L1BridgeSecure.sol):

- Accepts NPY deposits from L1
- Receives state roots from sequencer
- Verifies and processes withdrawal proofs

L2 Bridge (L2Bridge.sol):

- Mints L2 tokens on deposit confirmation
- Burns L2 tokens on withdrawal initiation
- Maintains Merkle tree of withdrawals

6.3 Withdrawal Flow

The withdrawal process follows these steps:

1. User calls `withdraw(amount)` on L2 Bridge
2. L2 records withdrawal in Merkle tree
3. Sequencer submits state root to L1 (every 10 blocks)
4. User generates Merkle proof via `turbo_getWithdrawalProof` RPC
5. User submits proof to L1 Bridge
6. L1 Bridge verifies proof and releases funds

7 Token Economics

7.1 NPY Token

Property	Value
Symbol	NPY
Decimals	18
Initial Supply	100,000,000 NPY
Block Reward	2 NPY
Block Time (L1)	12 seconds
Block Time (L2)	10 seconds

Table 4: Token Properties

7.2 Staking Economics

- **Minimum Stake**: 10,000 NPY
- **Lock Period**: 100 blocks (approximately 20 minutes)
- **Rewards**: Compound automatically to stake
- **Slashing**: 10% penalty for misbehavior

7.3 Gas Fees

- **L1 Gas Price**: 1 Gwei (0.000000001 NPY)
- **L2 Gas Price**: 1 Gwei
- **Typical Transfer**: 21,000 gas = 0.000021 NPY

Contract	Description
USDN	USD-pegged stablecoin
NanoPyDEX	AMM DEX (NPY/USDN)
NanoPyNFT	NFT collection
L1Bridge	L1-L2 bridge (L1 side)
L2Bridge	L1-L2 bridge (L2 side)

Table 5: Deployed Contracts

8 Smart Contracts

8.1 Core Contracts

8.2 NanoPyDEX

Uniswap-style constant product automated market maker:

$$x \cdot y = k \tag{2}$$

where x is the NPY reserve, y is the USDN reserve, and k is the constant product.

```
function swapNPYForUSDN(uint256 minUsdnOut) external payable;
function swapUSDNForNPY(uint256 usdnIn, uint256 minNpyOut) external;
function addLiquidity(uint256 usdnAmount) external payable;
function removeLiquidity(uint256 liquidityAmount) external;
```

Trading Fee: 0.3% per swap

9 Network Configuration

9.1 Testnet Configuration

Parameter	L1 Testnet	L2 Turbo
Chain ID	77777	777702
RPC URL	http://51.68.125.99:8546	http://51.68.125.99:8548
Symbol	NPY	NPY
Block Time	12s	10s

Table 6: Network Parameters

9.2 Developer Tools

- **Explorer**: nanopy-scan
- **Faucet**: nanopy-faucet (100 NPY per claim)
- **CLI**: `pip install nanopy-chain`

10 Development Guide

10.1 Installation

```
pip install nanopy-chain
```

10.2 Running a Node

```
nanopy-node --rpc-port 8545 --chain-id 77777
```

10.3 Running a Validator

```
nanopy-validator --validator-key YOUR_PRIVATE_KEY
```

10.4 Deploying Contracts

Configure Hardhat for NanoPy:

```
// hardhat.config.js
module.exports = {
  networks: {
    nanopy: {
      url: "http://51.68.125.99:8546",
      chainId: 77777,
      accounts: [PRIVATE_KEY]
    },
    turbo: {
      url: "http://51.68.125.99:8548",
      chainId: 777702,
      accounts: [PRIVATE_KEY]
    }
  }
};
```

11 Roadmap

11.1 Phase 1: Foundation (Complete)

- L1 blockchain with Proof of Stake
- Full EVM implementation
- JSON-RPC API
- P2P networking
- Block explorer
- Faucet

11.2 Phase 2: L2 Scaling (Complete)

- NanoPy Turbo sequencer
- L1-L2 bridge
- Withdrawal proofs
- State root submission

11.3 Phase 3: DeFi (In Progress)

- USDN stablecoin
- NanoPyDEX (AMM)
- NanoPySwap UI
- Lending protocol

11.4 Phase 4: Ecosystem

- Mainnet launch
- Token listing
- Developer grants
- Governance

12 Conclusion

NanoPy democratizes blockchain development by providing a complete, readable, and modifiable implementation in Python. With full Ethereum compatibility, Layer 2 scaling, and integrated DeFi primitives, NanoPy enables developers to learn, experiment, and build without the barriers of complex system languages.

The combination of an accessible Python L1 for education and experimentation, paired with a high-performance Go L2 for production workloads, creates a unique platform that serves both learning and real-world use cases.

NanoPy — Blockchain, in Python.

References

1. Wood, G. (2014). Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Yellow Paper*.
2. Buterin, V. et al. (2021). EIP-1559: Fee market change for ETH 1.0 chain. *Ethereum Improvement Proposals*.
3. Buterin, V. (2021). An Incomplete Guide to Rollups. *vitalik.ca*.
4. Protocol Labs. (2019). libp2p Specification. *libp2p.io*.