EVM Architecture and Smart Contract Engineering: Comprehensive Concepts

Lucas Kim
Loyola Marymount University
Los Angeles, US

Abstract

This study guide provides a structured, in-depth exploration of the Ethereum Virtual Machine (EVM), its architectural components, opcode mechanics, gas economics, security considerations, and extensions to tokenization standards. Designed for engineers preparing to build production-grade smart contracts and tokenized systems, the content emphasizes deterministic execution, state management, and practical implementation patterns. Forward-thinking insights highlight scalability via Layer 2 and hybrid computation models, encouraging consistent application of best practices to overcome perceived impossibilities in decentralized systems.

1 EVM Philosophy and Design Objectives

1.1 Blockchain Distributed Computing Challenge

Concept: The core problem in blockchain is trust in computation: "Who can be trusted to perform the calculation?"

Details: Centralized systems rely on a single authority that computes results and expects users to accept them without verification. Blockchains eliminate this by mandating that every participating node independently executes the same operations on identical inputs. This requires absolute determinism, where any deviation in hardware, software, or timing would produce different outputs and break consensus.

Notes: Bitcoin solved value transfer but limited computation to simple scripts. Ethereum extends this to general-purpose programming while preserving verifiability.

1.2 Limitations of Bitcoin Script

Concept: Bitcoin Script is intentionally non-Turing complete.

Details: It operates on a stack with basic opcodes but deliberately omits loops, recursion, and persistent state to guarantee termination and bounded resource use. Scripts are confined to validating spending conditions such as multisignature requirements or time-locked transactions. This design ensures safety and predictability but prevents complex program logic.

Notes: Safe for transactions but insufficient for decentralized applications requiring persistent state or conditional logic.

1.3 Ethereum Core Goal: Stateful Global Computer

Concept: Ethereum aims to create a "World Computer" with persistent, shared state.

Details: Smart contracts maintain internal variables that survive across multiple transactions and external calls, allowing autonomous behavior. Contracts can invoke each other composably, forming complex systems like decentralized exchanges or lending protocols. The entire network simulates a single replicated machine with a unified global state.

Notes: This shifts blockchain from a ledger to a computation platform, but introduces challenges in consensus and resource bounding.

1.4 Mathematical Interpretation of "World Computer"

Concept: Determinism plus consensus equals global verifiability.

Details: The state transition function $\Upsilon(\sigma, T) \rightarrow \sigma'$ takes the current world state σ and an ordered transaction list T to produce the next state σ'. Every honest node must compute the exact same σ' from the same inputs. Consensus mechanisms enforce agreement on T and punish deviations through slashing or chain reorganization.

Notes: Formalizes Ethereum as a replicated state machine; deviations cause forks, resolved via longest-chain or staking rules.

2 EVM Internal Architecture (Core Runtime Model)

2.1 Stack Component

Concept: 1024-slot limit, 256-bit words.

Details: The stack serves as the primary workspace for operands, operating in LIFO order with a hard cap of 1024 items to prevent unbounded recursion. Each item is a full 256-bit word, and arithmetic operations like ADD pop two values, compute the result modulo 2^{256}, and push the outcome. Underflow or overflow conditions immediately halt execution with an exception.

Notes: Depth limit prevents excessive recursion; visualize changes during execution for debugging.

2.2 Memory Component

Concept: Transient byte array, dynamically allocated.

Details: Memory starts at zero size and expands automatically when written via MSTORE or MSTORE8, with gas cost growing quadratically to penalize large allocations. It is byte-addressable and volatile, resetting completely at the end of each external call or transaction. This makes it ideal for temporary buffers during ABI encoding/decoding or cryptographic hashing.

Notes: Used for temporary data like function arguments; resets per invocation.

2.3 Storage Component

Concept: Persistent key-value store via Merkle Patricia Trie.

Details: Storage maps 256-bit keys to 256-bit values in a cryptographically committed trie structure, enabling efficient updates and Merkle proofs. SSTORE writes are expensive and create new trie nodes, while SLOAD reads access warmed or cold slots with tiered gas costs post-EIP-2929. Changes persist across transactions and are reflected in the account's storageRoot hash.

Notes: Expensive but durable; design contracts to minimize writes.

2.4 Program Counter, Gas, and Execution Context

Concept: PC tracks opcode offset; gas meters computation.

Details: The program counter advances one byte per opcode, while the gas meter decrements according to each operation's predefined cost to enforce resource limits. The execution context provides environmental variables like CALLER, ORIGIN, CALLVALUE, and BLOCKNUMBER for contract logic. Running out of gas triggers a safe revert that undoes state changes but preserves logs.

Notes: Gas exhaustion halts execution safely, solving the halting problem practically.

2.5 CALL Variants

Concept: CALL (new context, value transfer), DELEGATECALL (current context, no value), STATICCALL (read-only).

Details: CALL spawns a sub-context with its own msg.sender and isolated storage, allowing ETH transfer and full state modification. DELEGATECALL executes code in the caller's context, preserving msg.sender and storage while prohibiting value transfer. STATICCALL mirrors CALL but reverts on any state-modifying opcode like SSTORE.

Notes: DELEGATECALL enables proxies but risks storage collisions; use cautiously.

2.6 Opcode Classification

Concept: Grouped by function (Arithmetic, Stack, Environmental, etc.).

Details: Arithmetic opcodes perform modular math on 256-bit integers; stack opcodes manage items via PUSH, POP, DUP, and SWAP. Memory and storage opcodes handle transient and persistent data access, while control flow opcodes like JUMP and JUMPI enable branching. Environmental opcodes expose context information without side effects.

Notes: All operations on 256-bit integers; no floats to ensure determinism.

3 EVM Storage Structure (Merkle Patricia Trie)

3.1 Account State

Concept: Balance, Nonce, CodeHash, StorageRoot.

Details: External Owned Accounts hold only balance and nonce for transaction ordering and replay protection. Contract accounts extend this with codeHash (immutable keccak256 of bytecode) and storageRoot (root of the persistent trie). These four fields fully define an account's state in the world trie.

Notes: CodeHash ensures immutability post-deployment.

3.2 Storage Trie Mechanics

Concept: Key-value persistence with path compression.

Details: The Patricia trie encodes 256-bit keys as hex nibble paths, using extension nodes for shared prefixes and branch nodes for divergence. Leaf nodes store values, and every update generates a new root hash via incremental hashing. This structure balances space efficiency with cryptographic verifiability.

Notes: Enables light clients to verify data without full state.

3.3 Immutability and Verification

Concept: Block header chains to state root via Merkle proofs.

Details: A Merkle proof consists of sibling hashes along the path from a leaf to the root, allowing verification of inclusion or absence. Light clients check proofs against the trusted stateRoot in block headers. This mechanism underpins SPV security and cross-chain interactions.

Notes: Critical for SPV wallets and cross-chain bridges.

4 EVM Opcode Set

4.1 Arithmetic and Stack Operations

Concept: ADD, MUL, DIV, MOD, EXP; PUSHn, POP, DUPn, SWAPn.

Details: All arithmetic is performed modulo 2^{256} to prevent overflow exceptions, with EXP costing gas proportional to exponent bit length. PUSH opcodes load 1-32 byte constants onto the stack, while DUP and SWAP enable efficient item duplication and reordering. POP discards the top item without cost beyond gas accounting.

Notes: Stack visualization: Track pushes/pops to avoid underflow.

4.2 Memory, Storage, and Flow Control

Concept: MSTORE/MLOAD, SSTORE/SLOAD; JUMP/JUMPI, STOP/RETURN/REVERT.

Details: Memory opcodes read/write 32-byte words with offset alignment, while storage opcodes access the persistent trie with cold/warm access tiers. JUMPI conditionally alters the PC based on a stack condition, requiring a preceding JUMPDEST for validity. RETURN and REVERT end execution with data or error, respectively.

Notes: REVERT refunds remaining gas; use for error handling.

4.3 System and Environmental Opcodes

Concept: CALL/CREATE/SELFDESTRUCT, ADDRESS/CALLER/GAS.

Details: CREATE deploys new contract bytecode and returns its address, while CREATE2 adds a salt for deterministic addressing. LOG opcodes emit indexed topics for off-chain indexing, and environmental opcodes provide read-only context. SELFDESTRUCT removes code and sends balance but is restricted post-EIP-6780.

Notes: SELFDESTRUCT limited post-EIP-6780.

5 Gas System Mathematical Model

5.1 Gas Cost and Pricing

Concept: Opcode-specific costs (EIP-150, EIP-2929).

Details: Each opcode has a fixed base cost, with access costs for storage slots split into cold (first access) and warm (subsequent). EIP-1559 introduces a dynamic base fee burned per unit gas plus an optional priority tip to the block producer. Total transaction cost is GasUsed multiplied by (BaseFee + Tip).

Notes: EIP-1559 stabilizes fees; monitor for optimization.

5.2 Refunds and Limits

Concept: Refunds for SSTORE clearing or SELFDESTRUCT.

Details: Setting a storage slot to zero refunds a portion of the write cost, capped at 50**Notes:** Refunds bounded to prevent abuse; gas bounds Turing completeness.

6 Transaction Processing Pipeline

6.1 Step-by-Step Flow

Concept: 1. Tx creation/signature; 2. Broadcast; 3. Validation; 4. EVM execution; 5. State update; 6. Consensus.

Details: Transactions are signed with nonce to prevent replay, then propagated via mempool to validators. Validators execute the EVM in sequence, apply receipts, and update the world state trie. Full nodes replay the entire block to confirm the new state root matches.

Notes: Non-determinism (e.g., BLOCKHASH) bounded for safety.

7 Smart Contract Lifecycle

7.1 Compilation and Deployment

Concept: Solidity → Yul → Bytecode; CREATE for deployment.

Details: The Solidity compiler generates EVM bytecode via an intermediate Yul representation for optimization. Deployment uses CREATE to execute init code and set the contract address from sender and nonce. CREATE2 incorporates a salt for predictable addresses independent of nonce.

Notes: Use CREATE2 for predictable addresses.

7.2 Execution and Upgrade Patterns

Concept: CALL for invocation; Proxy (Transparent/UUPS) for upgrades.

Details: External calls use CALL with gas forwarding and value transfer, returning success and data. Proxy patterns delegate execution to upgradable logic contracts via DELEGATECALL, using standardized storage slots per EIP-1967. UUPS allows upgrade logic within the implementation itself.

Notes: Proxies separate logic/state; mitigate with access controls.

8 Smart Contract Language Structures

8.1 Solidity vs Vyper

Concept: Solidity (feature-rich, C-like); Vyper (secure, Python-like).

Details: Solidity supports inheritance, libraries, and inline assembly for fine control, while Vyper enforces explicitness and omits complex features to reduce bugs. Function selectors are the first four bytes of the keccak256 hash of the canonical signature. Events use LOG opcodes with indexed topics for filtering.

Notes: Vyper reduces attack surface; choose based on complexity.

8.2 Storage Packing and Modifiers

Concept: Pack variables ≤ 32 bytes per slot; modifiers for guards.

Details: Multiple variables fitting within 32 bytes share a storage slot to minimize SSTORE costs, with compiler handling layout. Modifiers wrap function logic for reusable checks like onlyOwner or nonReentrant. Receive and fallback functions handle plain ETH transfers.

Notes: Optimize packing to save gas.

9 EVM Computation Limits and Security

9.1 Halting and Reentrancy

Concept: Gas solves halting; DAO hack via recursive calls.

Details: The gas mechanism guarantees termination by exhausting resources before infinite loops, solving the halting problem practically. The DAO exploit involved re-entering a withdrawal function before updating balance via fallback. The Checks-Effects-Interactions pattern updates state before external calls to prevent this.

Notes: Use ReentrancyGuard from OpenZeppelin.

9.2 Overflow and Delegatecall Risks

Concept: SafeMath pre-0.8; storage clash in proxies.

Details: Before Solidity 0.8, arithmetic required explicit overflow checks via libraries like SafeMath. DELEGATECALL executes foreign code in the caller's storage context, risking slot collisions if layouts differ. EIP-150 adjusted gas forwarding to prevent call-stack attacks.

Notes: Audit delegatecall targets rigorously.

10 Computability and Turing Completeness

10.1 Bounded Turing Machine

Concept: EVM models Turing machine but gas-bounded.

Details: Without gas limits, the EVM could simulate any computable function via loops and conditionals on its infinite tape equivalents. Gas caps practical execution while preserving theoretical completeness for bounded inputs. Formal tools prove properties by modeling the state transition function.

Notes: Formal verification via SMT (Z3) or symbolic execution (Mythril).

11 Layer 2 and VM Extensions

11.1 EVM Compatibility

Concept: Equivalent (Arbitrum) vs compatible (Polygon).

Details: EVM-equivalent chains replicate bytecode execution exactly, enabling seamless migration. zkEVMs generate validity proofs

for state transitions, while alternative VMs like WASM expand language support. Rollups batch transactions off-chain and settle data on L1.

Notes: L2 reduces costs via rollups; compress calldata.

12 EVM and Real-World Computability

12.1 Oracle Integration

Concept: EVM is pure; oracles (Chainlink) feed external data.

Details: The EVM cannot access external networks natively, so oracles act as trusted bridges pushing verified data on-chain. Hybrid models combine deterministic on-chain logic with off-chain computation or randomness. Chainlink VRF provides provably fair randomness via commit-reveal.

Notes: Verify oracle decentralization to avoid single points.

13 EVM Immutability Guarantees

13.1 Code and State Mechanics

Concept: codeHash fixes bytecode; state append-only via blocks.

Details: The codeHash field commits to deployed bytecode immutably unless SELFDESTRUCT is invoked. State changes are appended as new trie versions linked through block headers. Consensus finality after epochs makes reversion economically infeasible.

Notes: Byzantine tolerance via deterministic ops.

14 Advanced Concepts

14.1 Yul and Tracing

Concept: Yul IR for optimization; traceTransaction for debugging.

Details: Yul is a low-level intermediate language allowing manual stack management and loop unrolling for gas savings. RPC trace methods expose per-opcode stack, memory, and storage changes. CREATE2 uses keccak256(0xff ++ sender ++ salt ++ initcode) for address prediction.

Notes: Profile gas; avoid refund exploits.

15 Token Standards and Asset Representation

15.1 ERC-20 (Fungible)

Concept: balanceOf, transfer, approve, allowance, transferFrom.

Details: The standard defines a uniform interface for fungible tokens with balance mapping and allowance for delegated spends. Total supply is tracked centrally, and transfers emit events for indexing. Solidity 0.8+ includes built-in overflow checks.

Notes: Standard for fungibles; extend with snapshots.

15.2 ERC-721 (Non-Fungible)

Concept: tokenID, metadata URI, ownerOf.

Details: Each token has a unique ID with ownership mapping and optional metadata pointer. Safe transfers verify receiver contracts implement the ERC721 interface to prevent loss. Approval can be per-token or operator-wide.

Notes: Ideal for unique assets; use enumerable for listings.

15.3 ERC-1155 (Multi-Asset)

Concept: Batch operations, ID-based types.

Details: A single contract manages multiple token types, where IDs distinguish fungible, semi-fungible, or NFT behavior. Batch transfer functions reduce gas for multi-item operations. Metadata can be shared or per-ID.

Notes: Combine fungible/NFT in one contract.

16 Tokenization Workflow on Ethereum

16.1 End-to-End Process

Concept: Registry → Issuance → Transfer → Redemption.

Details: Real-world assets are registered off-chain with legal wrappers, then minted as tokens via on-chain governance. Transfers follow standard interfaces, while redemption burns tokens and triggers off-chain settlement. Oracles or proofs validate reserve backing.

Notes: Proof-of-reserve via Merkle audits.

17 Economic Considerations in Contract Design

17.1 Optimization Patterns

Concept: Inline assembly, minimal proxies (EIP-1167).

Details: Assembly enables bit-level operations and custom gas savings beyond high-level compiler output. Minimal proxies deploy logic once and clone via CREATE2 for cheap instances. Batching multiple actions in one transaction amortizes fixed costs.

Notes: Prioritize gas; test under load.

18 Governance and Upgradeability

18.1 Proxy and Diamond Patterns

Concept: UUPS for flexible upgrades; EIP-2535 facets.

Details: UUPS places upgrade logic in the implementation contract, reducing proxy size. Diamond standard allows multiple facet contracts for modular functionality without storage conflicts. Governance typically uses token-weighted voting with timelocks.

Notes: Timelocks prevent rushed changes.

19 Legal and Compliance Considerations

19.1 Token Classification

Concept: Utility vs security; KYC on-chain.

Details: Tokens passing the Howey test are securities requiring regulatory compliance. On-chain KYC can use zero-knowledge proofs or allowlists via Merkle trees. Stablecoins under MiCA must maintain audited reserves.

Notes: Consult jurisdiction; build compliance hooks.

References

[1] G. Wood. Ethereum Yellow Paper. Available at https://ethereum.github.io/yellowpaper/paper.pdf.
[2] EIP-150: Gas cost changes. Available at https://eips.ethereum.org/EIPS/eip-150.
[3] OpenZeppelin Contracts. Available at https://docs.openzeppelin.com/contracts.