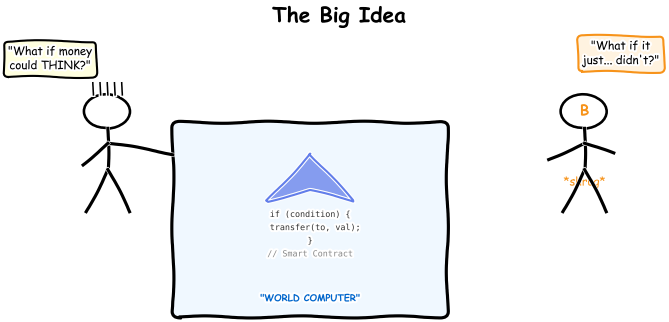
L05: Ethereum and the EVM – Technical Deep Dive

BSc Blockchain Course

Digital Finance

What If the Internet Could Run Programs Nobody Can Shut Down?



Bitcoin proved you can move money without a bank. Ethereum asks a bigger question: what if you could run *any program* on a global computer that no company, government, or individual can turn off? Today we open the hood of that machine and examine exactly how it works – from opcodes to gas to the world state trie.

This cartoon frames the central question of Lesson 5: can a blockchain be a general-purpose computer – and should it be?

Learning Objectives

By the end of this lesson you will be able to:

1. **Describe** Ethereum's account model and distinguish between externally owned accounts (EOAs) and contract accounts. *[Understand]*
2. **Explain** how the EVM executes bytecode using a stack-based architecture with deterministic gas metering. *[Understand]*
3. **Calculate** transaction costs under the EIP-1559 fee model (base fee, priority fee, and ETH burn). *[Apply]*
4. **Compare** Ethereum's account/state model with Bitcoin's UTXO model on expressiveness, privacy, and complexity. *[Analyze]*
5. **Evaluate** the trade-offs in Ethereum's roadmap decisions (The Merge, proto-danksharding, statelessness). *[Evaluate]*

Bloom's levels covered: Understand, Apply, Analyze, Evaluate

These objectives map directly to quiz and exercise assessments.

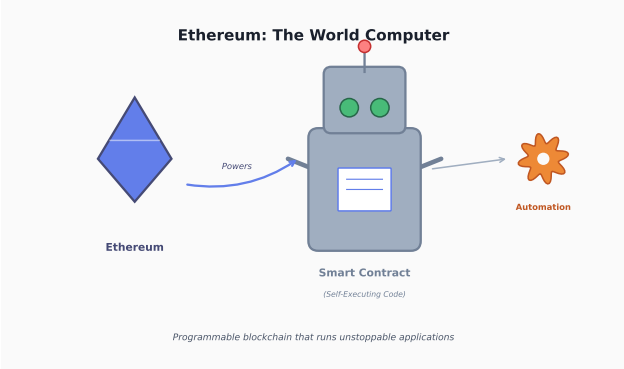
From Digital Cash to Programmable Money

In Lessons 1–4 we studied blockchain fundamentals, cryptography, consensus, and Bitcoin. Bitcoin answers one question: *can we transfer value without a bank?*

Ethereum asks a much broader question: *can we encode arbitrary agreements as self-executing code on a shared world computer?*

The leap from Bitcoin to Ethereum:

- **Bitcoin:** Limited scripting – can express "send 1 BTC to address X if condition Y" but little else.
- **Ethereum:** Turing-complete programming – can express any computable logic: lending, voting, games, identity.
- **Key insight:** Ethereum does not just move tokens; it transforms blockchain from a ledger into a platform.



- **What you see:** The evolution from simple value transfer to programmable smart contracts.
- **Key pattern:** Each generation of blockchain adds a new layer of expressiveness.
- **Takeaway:** Ethereum's power – and its complexity – stems from making blockchain Turing-complete.

The DAO Hack: When "Code Is Law" Met Reality

In June 2016 – barely a year after Ethereum launched – the community faced its first existential crisis.

What happened:

- **The DAO** (Decentralized Autonomous Organization) raised **$150 million** in ETH from 11,000 investors – the largest crowdfund in history at the time.
- An attacker discovered a **reentrancy bug**: the contract's withdraw function sent ETH *before* updating the balance, allowing recursive calls to drain $60 million.
- The community split: one side argued "the attacker followed the code's rules – no theft occurred." The other argued "the code had a bug – the community should fix it."

The outcome: Ethereum performed a **hard fork** to reverse the theft. Dissenters kept the unmodified chain – now called **Ethereum Classic (ETC)**.

Why this matters today: Every smart contract you write inherits this lesson. The EVM executes code *exactly as written* – bugs and all. Understanding the EVM is not optional; it is the difference between a working protocol and a $60 million exploit.

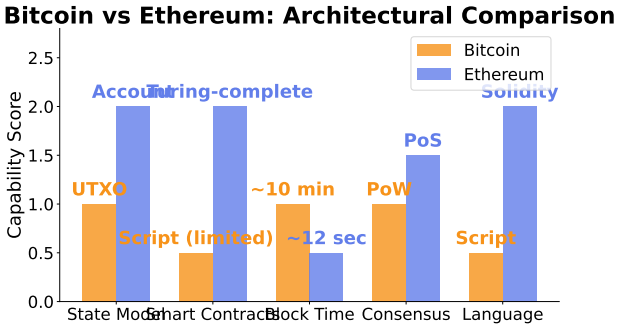
The DAO hack led directly to the development of formal verification tools and the "checks-effects-interactions" pattern.

Ethereum vs Bitcoin: Two Philosophies, One Technology

Bitcoin and Ethereum share a common ancestor (blockchain + consensus) but diverge on nearly every design decision.

Six fundamental differences:

- **Purpose:** Bitcoin = digital gold (store of value). Ethereum = world computer (general-purpose platform).
- **State model:** Bitcoin uses UTXOs (unspent transaction outputs). Ethereum uses accounts with balances.
- **Scripting:** Bitcoin Script is intentionally limited. Solidity/Vyper on Ethereum are Turing-complete.
- **Block time:** Bitcoin targets 10 minutes. Ethereum targets 12 seconds.
- **Supply:** Bitcoin has a hard cap of 21 million. Ethereum has no cap but burns fees (potentially deflationary).
- **Consensus:** Bitcoin uses Proof of Work. Ethereum switched to Proof of Stake in September 2022.



- **What you see:** A side-by-side comparison of Bitcoin and Ethereum across key technical dimensions.
- **Key pattern:** Bitcoin optimizes for simplicity and security; Ethereum optimizes for flexibility and expressiveness.
- **Takeaway:** Neither is "better" – they serve fundamentally different purposes.

Bitcoin's market cap is roughly 2x Ethereum's, but Ethereum settles more total value per day due to DeFi and stablecoins.

Ethereum's Vision: A World Computer

Vitalik Buterin's 2013 whitepaper proposed Ethereum as a "next-generation smart contract and decentralized application platform." The core idea:

The Ethereum Thesis

Any computation that can be expressed as a program should be expressible as a smart contract on a shared, censorship-resistant, globally replicated virtual machine.

Three design principles:

1. **Turing completeness:** The EVM can execute any algorithm, given enough gas. Loops, conditionals, and state storage are all supported – unlike Bitcoin's stack-only scripting.
2. **Account-based state:** Every address (user or contract) has a persistent state (balance, nonce, storage, code). The blockchain tracks the *current world state*, not just transaction history.
3. **Gas metering:** Every operation costs gas, preventing infinite loops and compensating validators. Gas is the EVM's resource accounting system.

Result: Ethereum became the platform for DeFi ($100B+ TVL), NFTs, DAOs, and thousands of tokens – all running on the same virtual machine.

"Turing-complete" means the system can simulate any Turing machine. In practice, gas limits bound execution to finite time.

Account Types: EOA vs Contract

Ethereum has exactly two types of accounts. Understanding their differences is essential for reasoning about transactions, gas, and security.

Externally Owned Account (EOA):

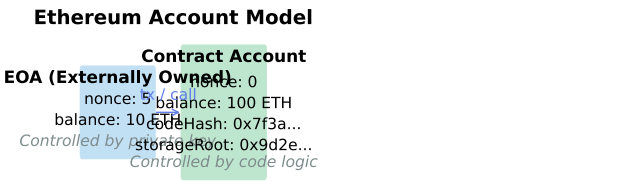
- Controlled by a private key (a human or bot)
- Can initiate transactions
- Has no code
- Address derived from public key
- Free to create (no on-chain cost)

Contract Account:

- Controlled by its deployed code
- Cannot initiate transactions – only responds to calls
- Has immutable bytecode and mutable storage
- Address derived from deployer address and nonce
- Costs gas to create (deploys bytecode on-chain)

Key trade-offs:

- **Security:** EOA = single key; contract = multi-sig, social recovery



- **What you see:** The two account types and their fields (nonce, balance, codeHash, storageRoot).
- **Key pattern:** EOAs have empty code/storage fields; contract accounts have all four populated.
- **Takeaway:** Every transaction on Ethereum originates from an EOA, even if it triggers a chain of contract calls.

Account State Fields: What the Blockchain Stores Per Account

Every Ethereum account – whether EOA or contract – is stored in the **world state** as a mapping from 160-bit address to four fields:

Field	Type	EOA	Contract Account
nonce	uint256	Number of transactions sent from this account	Number of contracts created by this contract
balance	uint256 (wei)	ETH held by this address (1 ETH = 10^{18} wei)	ETH held by the contract
codeHash	bytes32	Hash of empty string (`keccak256("")`)	Keccak-256 hash of the contract's immutable bytecode
storageRoot	bytes32	Hash of empty trie (no persistent storage)	Root of the account's storage trie (key-value store)

Key insights:

- The **nonce** prevents replay attacks: each transaction from an EOA must have a nonce exactly one higher than the last.
- The **balance** is denominated in wei (the smallest unit). 1 ETH = 10^{18} wei, similar to how 1 dollar = 100 cents.
- The **codeHash** is set once at deployment and can never change. This is what "immutable smart contracts" means.
- The **storageRoot** points to a Merkle Patricia Trie that holds all contract variables (mappings, arrays, structs).

The world state is a mapping: address → (nonce, balance, codeHash, storageRoot). All four fields fit in a single trie leaf.

Gas Mechanics: Why Every Instruction Has a Price

Gas is the unit of computational effort on Ethereum. Every EVM operation costs a fixed amount of gas, and every transaction must pay for the gas it consumes.

Why gas exists:

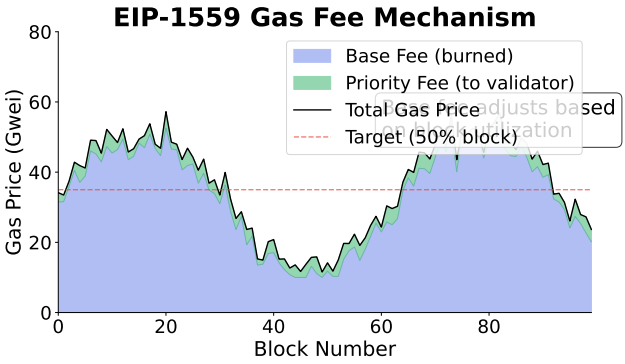
- **Halt problem:** Turing-complete programs can loop forever. Gas limits ensure every execution terminates.
- **Resource pricing:** Validators spend CPU, memory, storage, and bandwidth. Gas compensates them proportionally.
- **Spam prevention:** Without gas, an attacker could flood the network with infinite-loop contracts for free.

Gas formula:

$$\text{Fee} = \text{gasUsed} \times \text{gasPrice (gwei)}$$

Example: A simple ETH transfer uses 21,000 gas. At 30 gwei gas price with ETH at $2,000:

$$21{,}000 \times 30 \text{ gwei} = 0.00063 \text{ ETH} \approx \$1.26$$



- **What you see:** Gas costs for common operations, from simple transfers to complex contract interactions.
- **Key pattern:** Storage writes are the most expensive operations – 20,000 gas for a new slot vs 3 gas for an ADD.
- **Takeaway:** Gas-aware programming is not optional. Inefficient code costs users real money.

EIP-1559: Ethereum's Fee Market Revolution

Before August 2021, Ethereum used a first-price auction for gas: users bid blindly, and the highest bidder won. This led to overpaying, unpredictable fees, and MEV (Maximal Extractable Value) exploitation.

EIP-1559 replaced this with a two-component fee model:

Component	Description	Destination
Base fee	Algorithmically set by the protocol based on previous block utilization	**Burned** (destroyed forever) Reduces ETH supply
Priority fee (tip)	User-chosen incentive for validators to include the transaction faster	**Paid to the validator** who proposes the block

EIP-1559: Ethereum's Fee Market Revolution

Before August 2021, Ethereum used a first-price auction for gas: users bid blindly, and the highest bidder won. This led to overpaying, unpredictable fees, and MEV (Maximal Extractable Value) exploitation.

EIP-1559 replaced this with a two-component fee model:

Component	Description	Destination
Base fee	Algorithmically set by the protocol based on previous block utilization	**Burned** (destroyed forever) Reduces ETH supply
Priority fee (tip)	User-chosen incentive for validators to include the transaction faster	**Paid to the validator** who proposes the block

How the base fee adjusts:

- Target block utilization: 50% of the 30M gas limit (15M gas)
- If the previous block used **more** than 15M gas $\rightarrow$ base fee **increases** (up to 12.5% per block)
- If the previous block used **less** than 15M gas $\rightarrow$ base fee **decreases** (up to 12.5% per block)

Total cost:

$$\text{Fee} = \text{gasUsed} \times (\text{baseFee} + \text{priorityFee})$$

EIP-1559 was activated in the London hard fork (Aug 2021). Since then, over 4 million ETH has been burned.

EIP-1559 Fee Dynamics: A Worked Example

Problem: You want to swap tokens on Uniswap. The current base fee is 25 gwei. You set a priority fee of 2 gwei. The swap uses 150,000 gas. How much do you pay, and how much is burned?

Step 1 – Total gas price:

$$25 + 2 = 27 \text{ gwei}$$

EIP-1559 Fee Dynamics: A Worked Example

Problem: You want to swap tokens on Uniswap. The current base fee is 25 gwei. You set a priority fee of 2 gwei. The swap uses 150,000 gas. How much do you pay, and how much is burned?

Step 1 – Total gas price:

$$25 + 2 = 27 \text{ gwei}$$

Step 2 – Total fee:

$$150{,}000 \times 27 = 4{,}050{,}000 \text{ gwei}$$

$$= 0.00405 \text{ ETH} \approx \$8.10$$

EIP-1559 Fee Dynamics: A Worked Example

Problem: You want to swap tokens on Uniswap. The current base fee is 25 gwei. You set a priority fee of 2 gwei. The swap uses 150,000 gas. How much do you pay, and how much is burned?

Step 1 – Total gas price:

$$25 + 2 = 27 \text{ gwei}$$

Step 2 – Total fee:

$$150{,}000 \times 27 = 4{,}050{,}000 \text{ gwei}$$

$$= 0.00405 \text{ ETH} \approx \$8.10$$

Step 3 – Burned amount:

$$150{,}000 \times 25 = 3{,}750{,}000 \text{ gwei} = 0.00375 \text{ ETH}$$

Step 4 – Validator tip:

$$150{,}000 \times 2 = 300{,}000 \text{ gwei} = 0.0003 \text{ ETH}$$

EIP-1559 Fee Dynamics: A Worked Example

Problem: You want to swap tokens on Uniswap. The current base fee is 25 gwei. You set a priority fee of 2 gwei. The swap uses 150,000 gas. How much do you pay, and how much is burned?

Step 1 – Total gas price:

$$25 + 2 = 27 \text{ gwei}$$

Step 2 – Total fee:

$$150{,}000 \times 27 = 4{,}050{,}000 \text{ gwei}$$

$$= 0.00405 \text{ ETH} \approx \$8.10$$

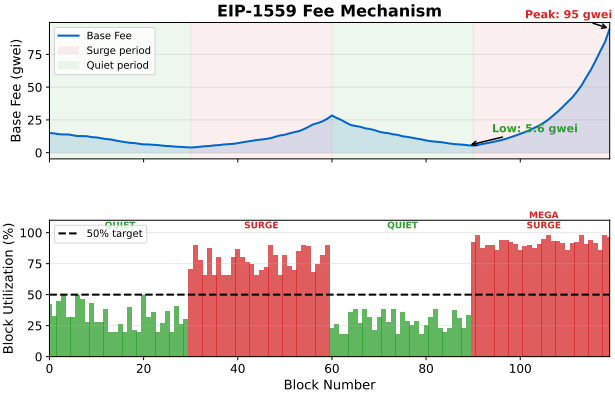
Step 3 – Burned amount:

$$150{,}000 \times 25 = 3{,}750{,}000 \text{ gwei} = 0.00375 \text{ ETH}$$

Step 4 – Validator tip:

$$150{,}000 \times 2 = 300{,}000 \text{ gwei} = 0.0003 \text{ ETH}$$

Result: Of your \$8.10 fee, **\$7.50 is burned** (removed from circulation) and **\$0.60 goes to the validator**.



- **What you see:** Base fee adjustments over time responding to block utilization above and below the 50% target.
- **Key pattern:** The base fee self-corrects – high demand pushes it up; low demand pulls it down.

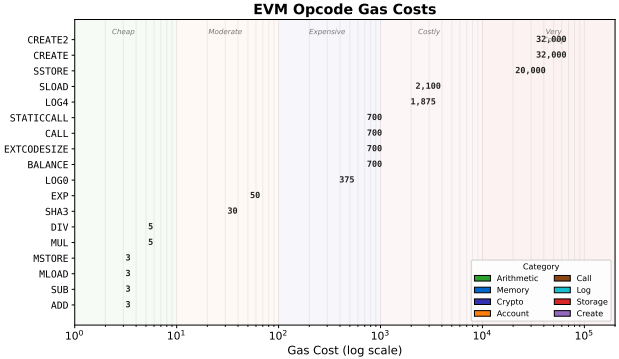
Gas Costs by Operation: What Makes Smart Contracts Expensive?

Not all EVM operations cost the same. Gas prices reflect the real computational and storage resources each operation consumes.

Gas cost categories:

1. **Arithmetic** (ADD, MUL, SUB): 3–5 gas. Cheap because they use only CPU.
2. **Memory access** (MLOAD, MSTORE): 3 gas base + expansion cost. Memory grows quadratically in cost.
3. **Storage writes** (SSTORE): 20,000 gas for a new slot; 5,000 gas to update. Expensive because storage persists forever.
4. **External calls** (CALL): 2,600+ gas. Cross-contract calls are expensive because they create new execution contexts.
5. **Contract creation** (CREATE): 32,000 gas base + 200 gas per byte of deployed code.

Optimization rule: Minimize storage writes. A single SSTORE costs as much as 6,600 ADDs.



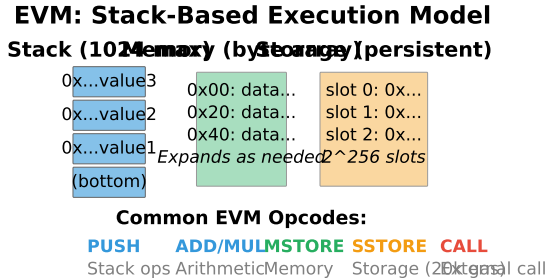
- **What you see:** Gas costs for common EVM opcodes, grouped by category.
- **Key pattern:** Storage operations dominate costs – a single SSTORE can cost more than an entire simple transaction.
- **Takeaway:** Gas-efficient Solidity requires understanding which opcodes your high-level code compiles to.

EVM Architecture: The Stack Machine Inside Ethereum

The **Ethereum Virtual Machine (EVM)** is a sandboxed, deterministic, stack-based virtual machine that executes smart contract bytecode.

Key architectural properties:

- **Stack-based:** Operands are pushed onto and popped from a 1,024-element stack. No registers.
- **256-bit word size:** Every stack element is 32 bytes, matching Keccak-256 hash output and Ethereum addresses.
- **Three data locations:**
  - **Stack:** Fast, temporary, max 1,024 depth
  - **Memory:** Byte-addressable, volatile, cleared after call
  - **Storage:** Key-value store, persistent, expensive
- **Deterministic:** Given the same state and input, every node produces the *identical* output – no randomness, no floating point, no system calls.



- **What you see:** The EVM execution model showing bytecode, stack, memory, and storage interactions.
- **Key pattern:** Bytecode instructions manipulate the stack; only SLOAD/SSTORE touch persistent storage.
- **Takeaway:** The EVM's simplicity (stack machine, no registers) makes formal verification feasible.

The EVM has roughly 140 opcodes. Compare this to x86 (1,500+ instructions) – simplicity is a design choice for security.

EVM Stack Operations: How Bytecode Executes

Let us trace a simple addition through the EVM to see how the stack machine works at the bytecode level.

Solidity code:

```
uint256 result = 3 + 5;
```

Compiled bytecode sequence:

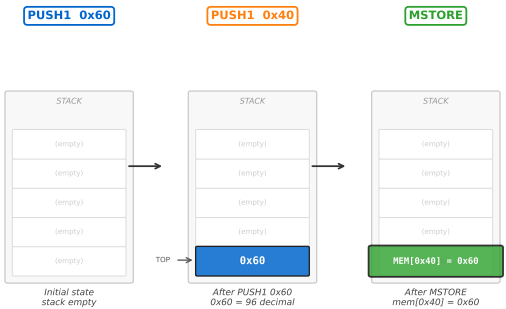
1. `PUSH1 0x03` – Push 3 onto stack [3]
2. `PUSH1 0x05` – Push 5 onto stack [5, 3]
3. `ADD` – Pop two, push sum [8]
4. `SSTORE` – Store 8 to a storage slot

Gas cost of this sequence:

- PUSH1: 3 gas (twice) = 6
- ADD: 3 gas
- SSTORE (new slot): 20,000 gas
- **Total: 20,009 gas**

The arithmetic (9 gas) is negligible. The storage write dominates.

EVM Stack: PUSH1 0x60 → PUSH1 0x40 → MSTORE



- **What you see:** Step-by-step stack state changes as opcodes execute sequentially.
- **Key pattern:** The stack is the EVM's only workspace – all computation flows through push/pop operations.
- **Takeaway:** Understanding opcodes helps debug gas costs, optimize contracts, and audit for vulnerabilities.

EVM Properties: Deterministic, Isolated, Metered

Three properties make the EVM suitable as a "world computer" that thousands of independent nodes can agree on:

Property	What It Means	Why It Matters
Deterministic	Same input + same state → same output No randomness, no floating point, no time-of-day	All nodes reach consensus on execution result. Without determinism, nodes would disagree.
Isolated (sandboxed)	Each contract call runs in its own execution context with its own stack, memory, and gas	A buggy contract cannot crash the EVM or access another contract's private storage.
Metered (gas)	Every opcode has a predetermined gas cost; execution halts when gas runs out	Infinite loops are impossible – they run out of gas. Validators are compensated fairly.

Consequences for developers:

- **No randomness:** You cannot call `random()` in a smart contract. Randomness requires an oracle (Chainlink VRF) or commit-reveal schemes.
- **No external I/O:** The EVM cannot fetch a web API, read a file, or access a database. External data enters only through transactions or oracle contracts.
- **No concurrency:** The EVM executes transactions sequentially within a block. Parallel execution is a research frontier (Monad, Sei).

The EVM's determinism means you can replay the entire Ethereum history from genesis and arrive at the exact same state.

The World State Trie: How Ethereum Stores Everything

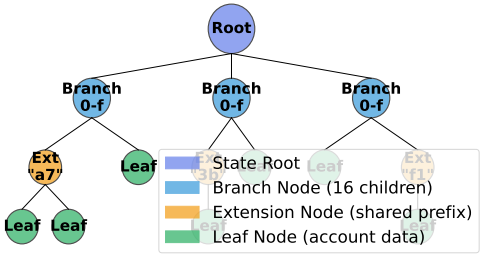
Ethereum's global state – every account's balance, nonce, code, and storage – is organized in a **Merkle Patricia Trie**, a data structure that combines three ideas:

1. **Trie (prefix tree):** Keys share common prefixes, enabling efficient lookup by address.
2. **Patricia:** Path compression – nodes with single children are merged to save space.
3. **Merkle:** Every node is hashed; the root hash summarizes the entire state in 32 bytes.

Three tries in every block:

- **State trie:** address → account data
- **Storage trie:** slot → value (per contract)
- **Transaction trie:** index → transaction

Ethereum State Trie (Modified Merkle Patricia Trie)



- **What you see:** The hierarchical structure of Ethereum's Merkle Patricia Trie with branch, extension, and leaf nodes.
- **Key pattern:** The stateRoot in the block header commits to the entire world state – change one balance and the root changes.
- **Takeaway:** Light clients can verify any account's state by downloading a short Merkle proof, not the full state.

Ethereum's full state is approximately 150 GB (2024). The Merkle Patricia Trie enables verification without storing all of it.

Merkle Patricia Trie: Why This Specific Data Structure?

Ethereum needed a data structure that satisfies four requirements simultaneously. No off-the-shelf solution existed, so a new one was designed.

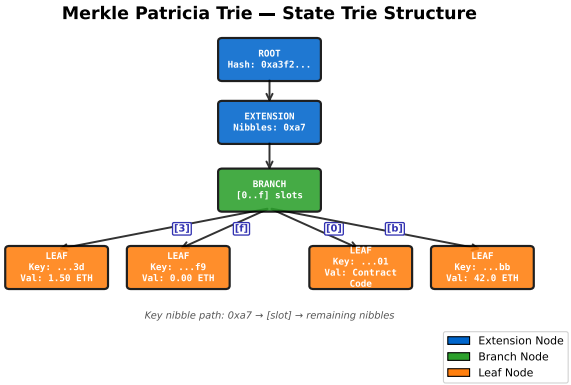
Four requirements:

1. **Efficient lookup:** Find any account's state in $O(\log n)$ time.
2. **Efficient update:** Modify one account without recomputing the entire tree.
3. **Cryptographic commitment:** A single root hash proves the integrity of all data below it.
4. **Proof of inclusion/exclusion:** Prove that an account exists (or does not) with a compact proof.

Why alternatives fail: A hash map gives $O(1)$ lookup but no cryptographic commitment. A plain Merkle tree gives commitments but requires fixed positions – inserting a new account changes every proof. **Only the Merkle Patricia Trie satisfies all four requirements.**

Node types:

- **Branch node:** 17 children (0–f hex digits + value)
- **Extension node:** Shared prefix + pointer to child
- **Leaf node:** Remaining key path + account data



- **What you see:** Detailed node types (branch, extension, leaf) and how keys are encoded in nibbles (hex digits).
- **Key pattern:** Path compression reduces tree depth – without it, every lookup would traverse 64 levels (one per hex digit of the address).

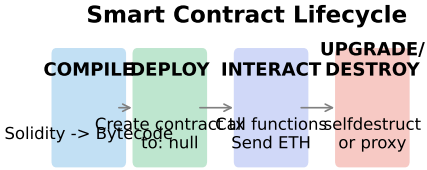
Smart Contract Lifecycle: Deploy, Execute, Destroy

A smart contract goes through a predictable lifecycle from creation to (optionally) removal. Each phase has distinct costs and risks.

Three phases:

1. **Deployment:** The creator sends a transaction with bytecode in the data field. The EVM executes the constructor, stores the runtime bytecode, and assigns a contract address.
2. **Execution:** Users call functions by sending transactions to the contract address. Each call consumes gas, reads/writes storage, and may call other contracts.
3. **Self-destruct** (deprecated): The `SELFDESTRUCT` opcode removes the contract's bytecode and sends remaining ETH to a specified address. Deprecated by EIP-6780 (Dencun, 2024).

Post EIP-6780: `SELFDESTRUCT` only works within the same transaction that deployed the contract. Existing contracts can no longer self-destruct.



Key Concepts:

- Constructor runs ONCE at deployment (initialization)
- Contract address = keccak256(deployer, nonce)
- Code is immutable after deployment (unless proxy pattern)
- selfdestruct sends remaining ETH to specified address

- **What you see:** The three lifecycle phases with gas costs and state changes at each stage.
- **Key pattern:** Deployment is the most expensive phase – bytecode storage costs 200 gas per byte.
- **Takeaway:** Once deployed, a contract's code is permanent (post EIP-6780). Bugs are forever.

The Ethereum Name Service (ENS) contract has been running continuously since May 2017 – nearly 8 years without downtime.

Anatomy of a Smart Contract: From Solidity to Bytecode

A smart contract written in Solidity goes through several transformations before executing on the EVM:

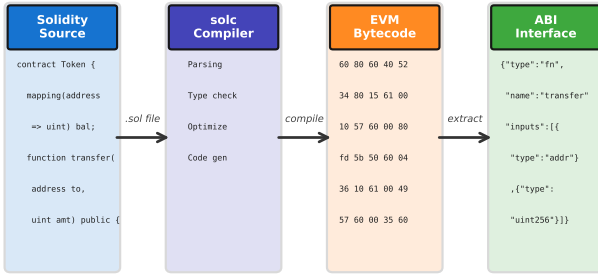
Compilation pipeline:

1. **Solidity source:** Human-readable code with functions, state variables, modifiers, and events.
2. **ABI (Application Binary Interface):** A JSON description of the contract's public functions. Used by frontends and other contracts to encode function calls.
3. **Bytecode:** The compiled output – a hex string of EVM opcodes. Split into:
   - *Creation bytecode:* Runs once during deployment
   - *Runtime bytecode:* Stored on-chain permanently

Contract anatomy:

- **State variables:** Stored in numbered slots (0, 1, 2, ...)
- **Functions:** Selected by the first 4 bytes of `keccak256(signature)`
- **Events:** Cheap logging mechanism (no storage, just logs)

Smart Contract Compilation Pipeline



Bytecode → deployed on-chain (EVM executes) | ABI → off-chain interface (dApps call)

- **What you see:** The pipeline from Solidity source through compilation to on-chain bytecode.
- **Key pattern:** The ABI acts as a "contract interface" – it tells callers which functions exist and what arguments they accept.

Contract Deployment: CREATE vs CREATE2

When deploying a smart contract, Ethereum offers two opcodes that determine how the contract's address is calculated:

CREATE (original):

- Address = rightmost 20 bytes of `keccak256(rlp(deployer, nonce))`
- Address depends on the deployer's nonce (transaction count)
- **Problem:** You cannot predict the address before deployment because the nonce may change

CREATE2 (EIP-1014, 2019):

- Address = rightmost 20 bytes of `keccak256(0xff, deployer, salt, keccak256(bytecode))`
- Address is determined by the deployer, a chosen salt, and the bytecode
- **Advantage:** Address is predictable before deployment – essential for counterfactual wallets, state channels, and cross-chain deployments

Use cases for CREATE2:

- Deploy the same contract to the same address on multiple chains
- Generate addresses for wallets that do not yet exist on-chain (counterfactual)
- Uniswap V2 uses CREATE2 to predict pair contract addresses without querying the factory

---

CREATE2 was introduced to support state channels and meta-transactions where contract addresses must be known in advance.

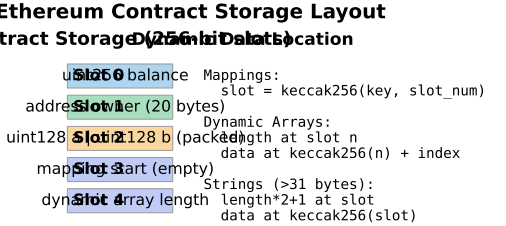
Storage Layout: How the EVM Organizes Contract Data

Every contract has 2^{256} storage slots, each holding 32 bytes. In practice, Solidity assigns variables to slots sequentially starting at slot 0.

Storage rules:

- **Fixed-size variables:** Assigned to slots in declaration order. Multiple variables smaller than 32 bytes can share a slot (packing).
- **Mappings:** Value for key k stored at slot `keccak256(k, p)` where p is the mapping's declared slot.
- **Dynamic arrays:** Length stored at slot p; elements start at `keccak256(p)`.
- **Strings/bytes:** Short values (<32 bytes) stored inline; longer values use the keccak256 pattern.

Security implication: Storage slots are deterministic. Attackers can read "private" variables directly from the state trie – the `private` keyword only prevents *other contracts* from reading the value, not off-chain observers.



Storage: Most expensive EVM operation (SSTORE = 20,000 gas for new value)

- **What you see:** How Solidity variables map to numbered storage slots, including packing of small types.
- **Key pattern:** Variables declared consecutively share slots when they fit within 32 bytes.
- **Takeaway:** Understanding storage layout is critical for gas optimization and proxy contract upgrades.

Slot packing saves 20,000 gas per avoided SSTORE. Ordering `uint128, uint128` uses one slot instead of two.

Storage Slot Packing: A Gas Optimization Technique

Problem: Which of these two contracts uses less gas for deployment and storage writes?

Contract A (unpacked):

```
uint256 a; // slot 0 (32 bytes)
uint256 b; // slot 1 (32 bytes)
uint256 c; // slot 2 (32 bytes)
```

Uses 3 storage slots.

Contract B (packed):

```
uint128 a; // slot 0, lower 16 bytes
uint128 b; // slot 0, upper 16 bytes
uint256 c; // slot 1 (32 bytes)
```

Uses 2 storage slots.

Storage Slot Packing: A Gas Optimization Technique

Problem: Which of these two contracts uses less gas for deployment and storage writes?

Contract A (unpacked):

```
uint256 a; // slot 0 (32 bytes)
uint256 b; // slot 1 (32 bytes)
uint256 c; // slot 2 (32 bytes)
```

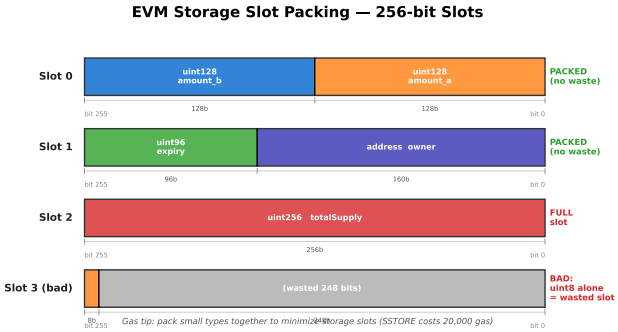
Uses 3 storage slots.

Contract B (packed):

```
uint128 a; // slot 0, lower 16 bytes
uint128 b; // slot 0, upper 16 bytes
uint256 c; // slot 1 (32 bytes)
```

Uses 2 storage slots. **Savings:**

- One fewer SSTORE per write: **20,000 gas saved**
- At 30 gwei + $2,000 ETH: **$1.20 saved per call**
- Over 1 million calls: **$1.2 million saved**



- **What you see:** Side-by-side comparison of packed vs unpacked storage layouts with gas costs.
- **Key pattern:** Grouping small types together reduces the number of storage slots touched.
- **Takeaway:** Variable ordering is a free optimization – reordering declarations costs nothing but saves gas.

Solidity packs only within a single slot. A `uint128` followed by a `uint256` uses two slots – the uint256 cannot split.

Ethereum Transaction Types

Ethereum has evolved from a single transaction format to multiple types, each serving a different purpose.

Type 0 – Legacy (pre-EIP-2718):

- Simple gas price auction
- Fields: nonce, gasPrice, gasLimit, to, value, data

Type 1 – Access List (EIP-2930):

- Declares which storage slots will be accessed
- Reduces cold-access gas penalties

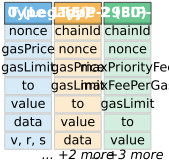
Type 2 – EIP-1559 (London fork):

- Replaces gasPrice with maxFeePerGas + maxPriorityFeePerGas
- Most common type today (90%+ of transactions)

Type 3 – Blob (EIP-4844, Dencun):

- Carries large data "blobs" for L2 rollups
- Separate blob gas market with its own base fee
- Reduces L2 data costs by 10–100x

Ethereum Transaction Types



Type 2 (EIP-1559) is now standard - enables predictable fees and ETH burning

accessList pre-declares storage slots for gas savings

- **What you see:** The four transaction types with their fields and when each was introduced.
- **Key pattern:** Each new type adds capabilities without breaking backward compatibility.
- **Takeaway:** Type 2 (EIP-1559) is the default; Type 3 (blobs) is transforming L2 economics.

EIP-4844 (proto-danksharding) reduced Arbitrum and Optimism fees from $0.50 to under $0.01 per transaction.

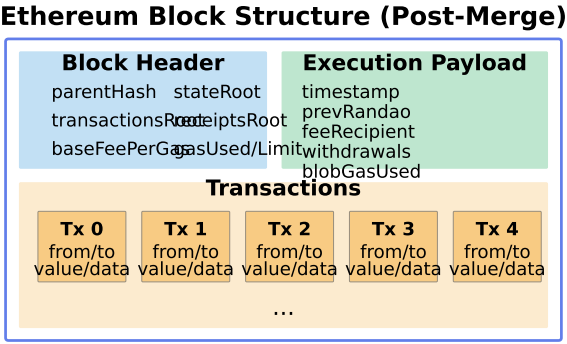
Ethereum Block Structure: What a Validator Proposes

An Ethereum block (post-Merge) contains two major components: the **consensus layer** data and the **execution layer** payload.

Block header fields:

- **parentHash:** Hash of the previous block
- **stateRoot:** Root of the world state trie after executing all transactions in this block
- **transactionsRoot:** Merkle root of all transactions
- **receiptsRoot:** Merkle root of all receipts (gas used, logs, status)
- **baseFeePerGas:** Current base fee (EIP-1559)
- **gasUsed / gasLimit:** Actual vs maximum gas
- **timestamp:** Unix timestamp of block creation
- **withdrawalsRoot:** Root of validator withdrawals (post-Shanghai)

Key difference from Bitcoin: Ethereum blocks commit to the *state* (via stateRoot), not just transactions.



- **What you see:** The anatomy of a post-Merge Ethereum block with consensus and execution layer fields.
- **Key pattern:** Three trie roots (state, transactions, receipts) commit to different aspects of the block.
- **Takeaway:** The stateRoot enables light clients to verify any account balance with a single Merkle proof.

ETH Issuance vs Burn: Is Ethereum Deflationary?

Since The Merge (Sept 2022) and EIP-1559, Ethereum has two competing forces on ETH supply:

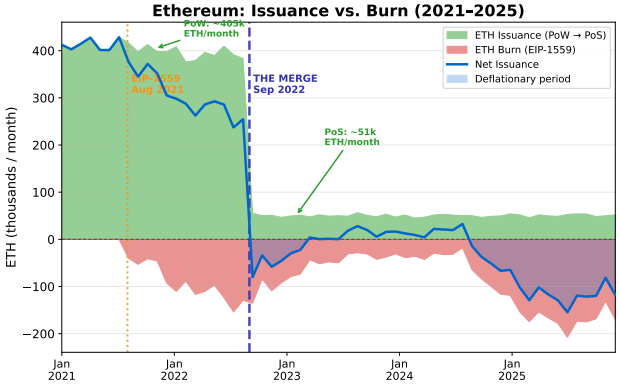
Supply increase (issuance):

- Validators receive new ETH for proposing and attesting to blocks
- Current issuance: approximately 0.5% per year (roughly 1,600 ETH/day)
- Much lower than pre-Merge PoW issuance (approximately 13,000 ETH/day)

Supply decrease (burn):

- EIP-1559 burns the base fee of every transaction
- Burn rate depends on network activity
- On busy days: burn exceeds issuance → net deflationary
- On quiet days: issuance exceeds burn → net inflationary

Net effect (2023–2024): ETH supply has been roughly flat – hovering around 120.2 million ETH, compared to Bitcoin's monotonically increasing supply.



- **What you see:** Daily issuance vs daily burn, showing periods of net inflation and net deflation.
- **Key pattern:** High-activity periods (NFT mints, DeFi surges) push burn above issuance, making ETH deflationary.
- **Takeaway:** ETH's monetary policy is "minimum viable issuance"

The Merge and the Ethereum Roadmap

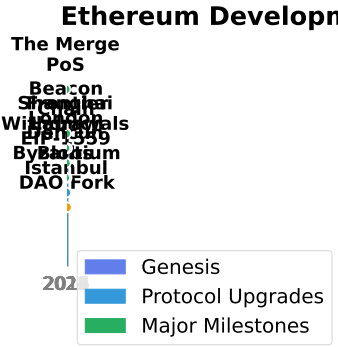
On 15 September 2022, Ethereum switched from Proof of Work to Proof of Stake – the most significant upgrade in blockchain history.

The Merge in numbers:

- **Energy:** Reduced by 99.95% – from Finland-scale to a small village
- **Validators:** 500,000+ staking 32 ETH each; slashing punishes misbehavior
- **Issuance:** Dropped 88% (13,000 → 1,600 ETH/day)
- **Finality:** 12.8 minutes (2 epochs × 32 slots × 12s)

Five roadmap workstreams:

1. **The Surge:** Scalability via rollups and data availability. EIP-4844 targets 100,000+ TPS across L2s.
2. **The Scourge:** Mitigate MEV and censorship via proposer-builder separation (PBS).
3. **The Verge:** Stateless clients via Verkle Trees (EIP-6800).
4. **The Purge:** Reduce state bloat; expire old data (EIP-4444).



- **What you see:** The five roadmap workstreams with their key milestones and target dates.
- **Key pattern:** Each workstream addresses a different axis of the scalability/decentralization/security trilemma.
- **Takeaway:** Ethereum's strategy is "rollup-centric" – L1 becomes the settlement and data availability layer; L2s handle execution.

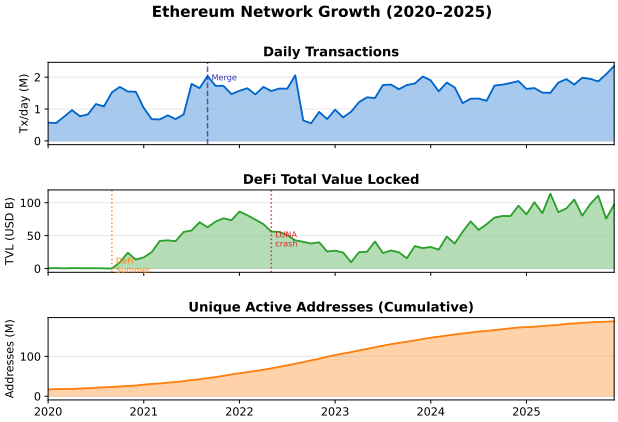
Ethereum by the Numbers: Network Statistics

How does Ethereum measure up as a global computing platform? Key metrics reveal both its achievements and its limitations.

Network metrics (2024):

- **Active validators:** approximately 900,000 (staking 29+ million ETH)
- **Daily transactions:** approximately 1.1 million (L1 only)
- **Total value secured:** approximately $400 billion (ETH + ERC-20 tokens + DeFi TVL)
- **Smart contracts deployed:** approximately 60 million (cumulative)
- **L2 transactions:** approximately 10–15 million/day (Arbitrum, Optimism, Base, zkSync combined)
- **Average gas price:** 15–40 gwei on L1
- **State size:** approximately 150 GB (growing at roughly 20 GB/year)

Trend: L2 transaction count now exceeds L1 by 10x – the rollup-centric roadmap is working.



- **What you see:** Key Ethereum network metrics over time – validators, transactions, and value secured.
- **Key pattern:** Validator count and L2 activity are growing; L1

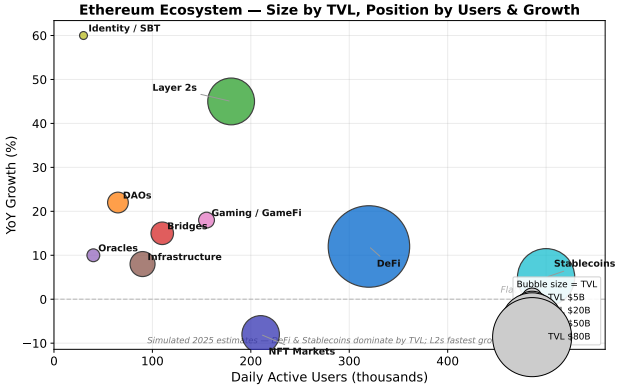
The Ethereum Ecosystem: Protocols, Tools, and Infrastructure

Ethereum is not just a blockchain – it is an **ecosystem** of interdependent protocols, developer tools, and infrastructure providers.

Ecosystem layers:

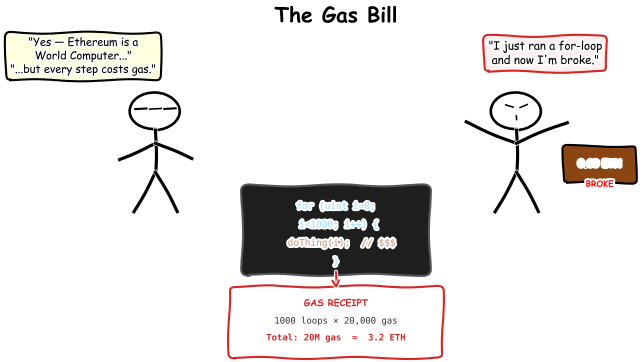
- **Infrastructure:** Execution clients (Geth, Nethermind), consensus clients (Prysm, Lighthouse), RPC providers (Alchemy, Infura)
- **Developer tools:** Hardhat, Foundry, Remix IDE, OpenZeppelin libraries, formal verification (Certora, Halmos)
- **DeFi:** Uniswap, Aave, MakerDAO, Lido, Curve – over $100 billion in TVL
- **L2 rollups:** Arbitrum, Optimism, Base, zkSync, StarkNet – processing 10x more transactions than L1
- **Identity and social:** ENS (Ethereum Name Service), Lens Protocol, Farcaster

Network effect: Ethereum's ecosystem is its strongest moat. Developers, users, and capital create a self-reinforcing cycle that competitors struggle to replicate.



- **What you see:** A map of the Ethereum ecosystem showing major protocols, tools, and infrastructure across categories.
- **Key pattern:** The ecosystem is broad and deep – no other blockchain comes close in developer activity.

One Last Thought...



Now you understand what happens beneath the surface when you interact with a smart contract – from the Solidity source code, through the compiler, into EVM bytecode, onto the stack, and finally into the world state trie. The machine is complex, but its rules are transparent and verifiable by anyone.

The best Ethereum developers start by understanding the EVM, not by memorizing Solidity syntax.

Key Takeaways

1. **Ethereum defined:** A Turing-complete blockchain that extends Bitcoin's value-transfer model into a general-purpose computing platform via the EVM.
2. **Account model:** Two account types (EOA and contract) with four state fields (nonce, balance, codeHash, storageRoot) stored in a Merkle Patricia Trie.
3. **Gas mechanics:** Every EVM operation has a gas cost. Storage writes (20,000 gas) dominate costs. EIP-1559 burns the base fee and pays validators a priority tip.
4. **EVM execution:** A deterministic, sandboxed, stack-based virtual machine with 256-bit words, three data locations (stack, memory, storage), and approximately 140 opcodes.
5. **The Merge:** Ethereum's switch from PoW to PoS (Sept 2022) cut energy by 99.95% and reduced ETH issuance by 88%.
6. **Roadmap:** Five workstreams (Surge, Scourge, Verge, Purge, Splurge) aim to make Ethereum scalable, censorship-resistant, and accessible to light clients – all centered on a rollup-centric architecture.

Review question: Calculate the total fee for a 200,000-gas transaction with base fee 20 gwei and priority fee 3 gwei. How much is burned?

Summary / Next Lesson Preview

Ethereum extends Bitcoin from a value-transfer ledger to a general-purpose computing platform via the EVM – a deterministic, sandboxed, gas-metered stack machine that executes smart contracts. The account model, Merkle Patricia Trie state storage, and EIP-1559 fee market form the technical foundations. The Merge to Proof of Stake and the rollup-centric roadmap chart Ethereum's path toward scalability without sacrificing decentralization.

Key Vocabulary:

- Ethereum Virtual Machine (EVM)
- EOA vs Contract Account
- Gas / EIP-1559 / Base Fee
- Merkle Patricia Trie
- The Merge (PoW to PoS)
- L2 Rollups

Next lesson: *L06: Solidity* – Writing code that handles other people's money requires more than just getting the logic right.

Review: Calculate the total fee for a 200,000-gas transaction with base fee 20 gwei and priority fee 3 gwei. How much is burned?