Ethereum & Smart Contracts: A Quantitative Deep Dive

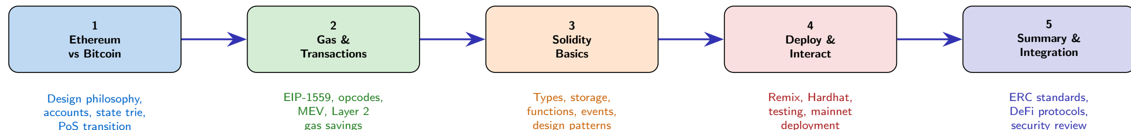
Standalone Technical Lecture

Prof. Dr. Joerg Osterrieder

University Lecture Series

March 5, 2026

Lecture Roadmap



Central Question

How does Ethereum extend blockchain from a payment ledger to a general-purpose decentralized computation platform?

Learning Objectives

- Contrast Ethereum's account model with Bitcoin's UTXO model
- Analyze gas economics and EIP-1559 fee markets
- Write, deploy, and test Solidity smart contracts
- Evaluate common vulnerabilities and design patterns

Prerequisites

- Blockchain fundamentals (hashing, Merkle trees, consensus)
- Basic programming experience
- Familiarity with Bitcoin transaction model

5

sections, ~60 frames covering Ethereum architecture, Solidity, deployment, and DeFi integration

Learning Objectives

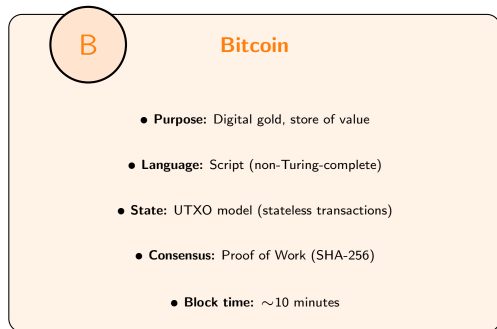
By the end of this lecture, you will be able to:

1. **Compare** Ethereum's account model with Bitcoin's UTXO model
2. **Calculate** gas costs for transactions using the EIP-1559 fee market mechanism
3. **Implement** basic Solidity contracts with state variables, functions, and modifiers
4. **Analyze** smart contract security vulnerabilities (reentrancy, overflow, access control)
5. **Evaluate** upgradeability patterns and their trust implications

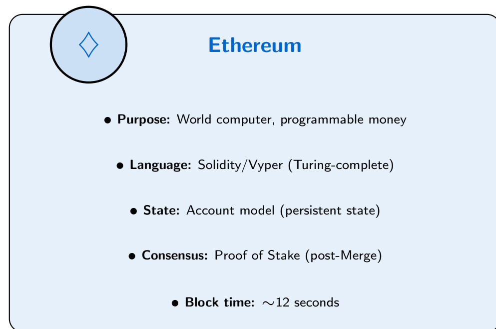
---

Bloom taxonomy levels: Remember → Understand → Apply → Analyze → Evaluate → Create

Bitcoin vs Ethereum: Design Philosophy



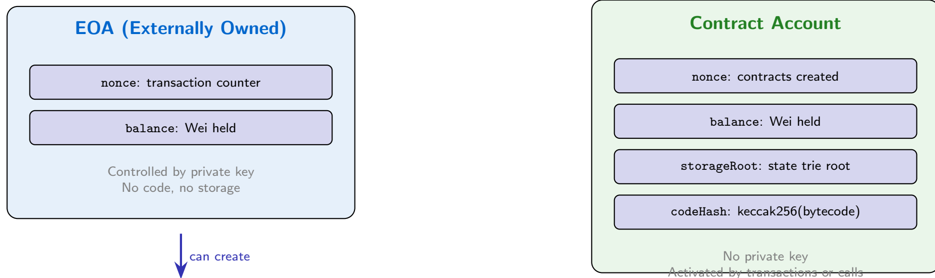
Paradigm Shift



Vitali

Buterin (2013): "Bitcoin is a calculator; Ethereum is a smartphone" – general-purpose vs. single-purpose

Account Model Deep Dive

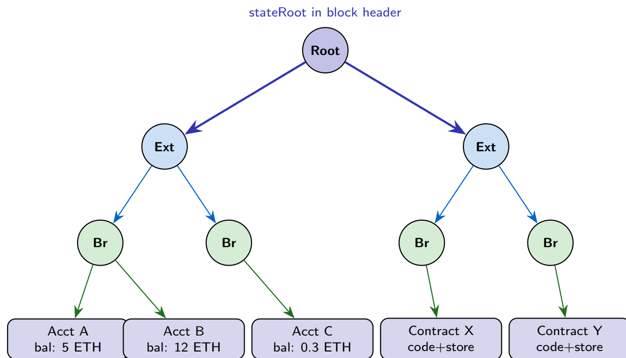


Key Differences from Bitcoin's UTXO Model

- **UTXO**: Each "coin" is a discrete, unspent output consumed entirely – naturally parallel but stateless
- **Account**: Global state maps addresses to balances/storage – enables complex state transitions but requires sequential nonce ordering
- **Trade-off**: Account model simplifies smart contract programming at the cost of reduced parallelism

Every Ethereum address is 20 bytes (160 bits) derived from keccak256 of the public key

State Trie Architecture



Modified Merkle Patricia Trie

- **3 node types**: Extension, Branch (16 children), Leaf
- Keys are nibble paths (hex characters of address)
- Path compression via extension nodes
- Every state change produces a **new root hash**

Four Tries per Block

1. **State trie**: all account data
2. **Storage trie**: per-contract storage (nested)
3. **Transaction trie**: block transactions
4. **Receipt trie**: execution results

Proof of Inclusion

Any account state is verifiable with $\mathcal{O}(\log n)$ hashes from the root – no need to download full state.

Ether

state trie currently stores >250M accounts; each block header commits to the entire world state

Ethereum Block Structure

Ethereum Block Header

- `parentHash`: previous block
- `stateRoot`: world state trie
- `transactionsRoot`: tx trie
- `receiptsRoot`: receipt trie
- `logsBloom`: 256-byte filter
- `baseFeePerGas`: EIP-1559
- `gasLimit`: 30M target
- `gasUsed`: actual consumed
- `timestamp`: Unix epoch
- `number`: block height

Bitcoin vs Ethereum Header

Field	BTC	ETH
Parent hash	✓	✓
Merkle root	1 (txs)	3 tries
Nonce/PoW	✓	– (PoS)
State root	–	✓
Gas fields	–	✓
Logs bloom	–	✓
Difficulty	✓	– (PoS)

Post-Merge Additions

- `withdrawalsRoot`: validator withdrawals
- `blobGasUsed`: EIP-4844 blobs (Dencun)
- `parentBeaconBlockRoot`: consensus layer link

Key Insight

Ethereum's header commits to the **entire world state**, not just transactions – enabling light client state proofs.

Transaction Types

	Type 0 (Legacy)	Type 1 (EIP-2930)	Type 2 (EIP-1559)
EIP	Pre-Berlin	EIP-2930	EIP-1559
Fee model	Single `gasPrice`	Single `gasPrice`	`baseFee` + `priorityFee`
Access list	–	✓	✓
Fee burning	–	–	✓ (base fee burned)
Key fields	`nonce, to, value, gasPrice, gas, data, v, r, s`	+ `accessList [addr, slots]`	+ `maxFeePerGas, maxPriorityFee, accessList`
Gas savings	Baseline	Warm storage discount	Predictable pricing

Access Lists (Type 1+)

- Pre-declare storage slots and addresses you will access
- First access to cold slot: 2100 gas → 100 gas (warm)
- Prevents surprise gas costs from `SLOAD` repricing

EIP-1559 Advantage (Type 2)

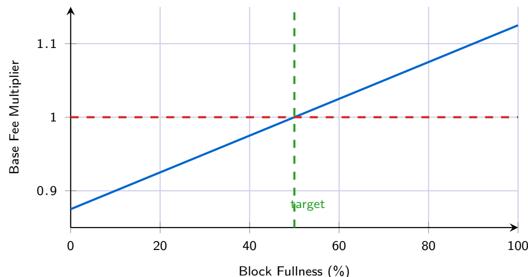
- Users set **max willingness to pay**, not exact price
- Base fee auto-adjusts – no more fee guessing
- Overpayment is **refunded**, not captured by validator
- Base fee is burned → deflationary pressure on ETH

As of 2024, ¿95% of Ethereum transactions use Type 2 (EIP-1559) for predictable fee estimation

EIP-1559 Fee Market Mechanism

Base Fee Adjustment Formula

$$\text{baseFee}_{\text{new}} = \text{baseFee}_{\text{old}} \times \left(1 + \frac{1}{8} \times \frac{\text{gasUsed} - \text{target}}{\text{target}}\right)$$



Mechanism Properties

- **Target**: 15M gas (50% of 30M limit)
- **Max change**: $\pm$12.5% per block
- Block <50% full $\rightarrow$ base fee **decreases**
- Block >50% full $\rightarrow$ base fee **increases**
- Empty block: fee drops by 12.5%
- Full block: fee rises by 12.5%

User Pays

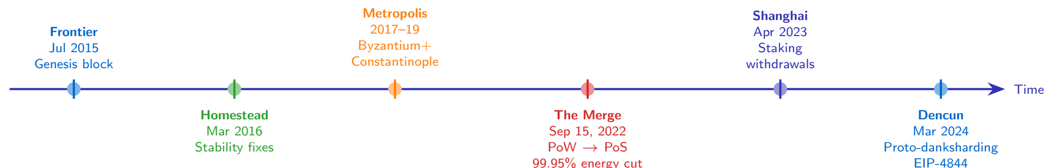
$$\text{fee} = \min(\texttt{maxFee}, \ \texttt{baseFee} + \texttt{tip})$$

- `baseFee` $\rightarrow$ **burned** (destroyed)
- `priorityFee` (tip) $\rightarrow$ validator
- Excess refunded to sender

Deflationary Effect

Since Aug 2021: >4M ETH burned. During high activity

The Merge: PoW to PoS



Before the Merge (PoW)

- Miners solve SHA-3/Ethash puzzles
- ∼15 TH/s network hashrate
- Energy: ∼78 TWh/year (comparable to Chile)
- Block reward: 2 ETH + fees
- Issuance: ∼13,000 ETH/day

After the Merge (PoS)

- Validators stake 32 ETH as collateral
- Randomly selected to propose blocks
- Energy: ∼0.01 TWh/year (**99.95% reduction**)
- Issuance: ∼1,600 ETH/day
- Net issuance often **negative** (deflationary)

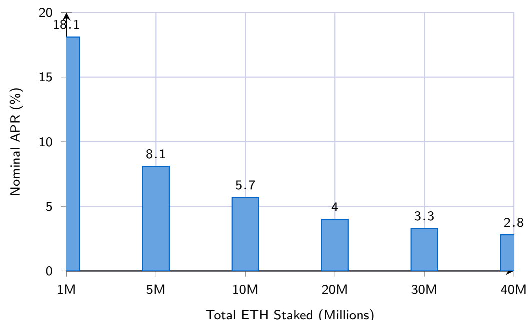
Technical Achievement

The Merge was the largest live infrastructure migration in crypto history – swapping consensus on a $200B+ network with zero downtime.

Beac

Chain launched Dec 2020; merged with execution layer Sep 15, 2022 at TTD = 58,750,000,000,000,000,000,000

Validator Economics



Staking Requirements

Parameter	Value
Min stake	32 ETH
Activation queue	~1–40 days
Withdrawal delay	~1–5 days
Max validators	~1M (soft)
Epoch length	32 slots (6.4 min)
Slot time	12 seconds

Slashing Conditions

- **Double voting**: proposing 2 blocks in same slot
- **Surround voting**: contradictory attestations
- Penalty: 1/32 of stake initially, up to full stake
- Correlated slashing: penalty increases if many slash simultaneously

Reward Sources

Validators earn from: (1) attestation rewards, (2) block proposal rewards, (3) sync committee duties, (4) MEV tips via mev-boost. Total APR ≈ 4–5% at 30M ETH staked.

Ethereum vs Bitcoin: Comprehensive Comparison

Dimension	Bitcoin	Ethereum
Consensus	Proof of Work (SHA-256)	Proof of Stake (Casper FFG)
Block time	$\sim$10 minutes	$\sim$12 seconds
Finality	Probabilistic (6 conf $\approx$ 60 min)	Economic ($\sim$13 min, 2 epochs)
State model	UTXO (stateless)	Account (stateful)
Smart contracts	Limited (Script)	Turing-complete (EVM)
Programming	Script (stack-based, $\sim$100 opcodes)	Solidity/Vyper ($\sim$140 opcodes)
Supply cap	21M BTC (hard cap)	No hard cap (net issuance $\approx$ 0)
Fee model	Fee market auction	EIP-1559 (base + tip, burning)
Throughput	$\sim$7 TPS	$\sim$15–30 TPS (L1)
Energy use	$\sim$150 TWh/yr	$\sim$0.01 TWh/yr
Native token	BTC (divisible to satoshi)	ETH (divisible to Wei, 10^{-18})
Upgrades	BIPs, conservative	EIPs, more frequent forks
L2 scaling	Lightning Network (channels)	Rollups (Optimistic, ZK)
Primary use	Store of value, payments	DeFi, NFTs, DAOs, dApps

Key Takeaway

Bitcoin optimizes for **security and simplicity** (sound money); Ethereum optimizes for **expressiveness and composability** (programmable finance). They serve complementary roles in the ecosystem.

Both networks have ¿99.9% uptime since launch – a remarkable achievement for decentralized systems

Gas: The Execution Fuel

Car Analogy

Fuel (liters) ↔ **Gas (units)**
Price per liter ↔ **Gas price (Gwei)**
Tank capacity ↔ **Gas limit**
Total cost ↔ **Gas used** × **price**

Why Gas Exists

- Prevents infinite loops (halting problem)
- Allocates scarce computation fairly
- Compensates validators for work
- Creates market for block space

Gas Mechanics

- Every EVM opcode has a fixed gas cost
- Transaction specifies a `gasLimit` (max gas to use)
- If execution exceeds limit → **revert** (state unchanged)
- Unused gas is **refunded** to sender
- Gas is **not** a token – it is denominated in ETH

Units

Unit	Wei	Use
Wei	1	Smallest unit
Gwei	10^9	Gas prices
Ether	10^{18}	Account balance

Turing Tax

Gas is the "Turing tax" – the cost of arbitrary computation on a replicated state machine. Every node re-executes every transaction.

A

EVM Opcode Gas Costs

Category	Opcode	Gas
Arithmetic		
	ADD / SUB	3
	MUL / DIV	5
	ADDMOD / MULMOD	8
	EXP	10+
Hashing		
	KECCAK256 (base)	30
	+ per word	6
Memory		
	MLOAD / MSTORE	3
	MSTORE8	3

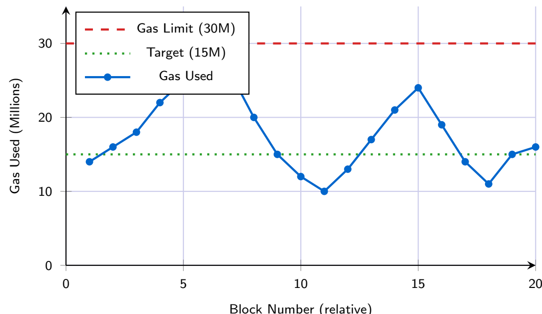
Category	Opcode	Gas
Storage (most expensive)		
	SLOAD (cold)	2,100
	SLOAD (warm)	100
	SSTORE (new)	20,000
	SSTORE (update)	5,000
Contract Creation		
	CREATE	32,000
	CREATE2	32,000
Calls		
	CALL (cold)	2,600
	CALL (warm)	100
	DELEGATECALL	2,600

Key Insight: Storage Dominates Cost

Storage operations are 100–6,600× more expensive than arithmetic. A single `SSTORE` to a new slot (20,000 gas) costs as much as 6,667 additions. **Optimizing storage access is the primary lever for gas reduction.**

Gas costs were recalibrated in EIP-2929 (Berlin, Apr 2021) introducing cold/warm access pricing

Gas Limit & Block Space



Block Space Economics

- **Gas limit**: 30M gas (hard ceiling)
- **Target**: 15M gas (50% of limit)
- Blocks can temporarily burst to 2× target
- EIP-1559 adjusts base fee to converge on target

What Fits in a Block?

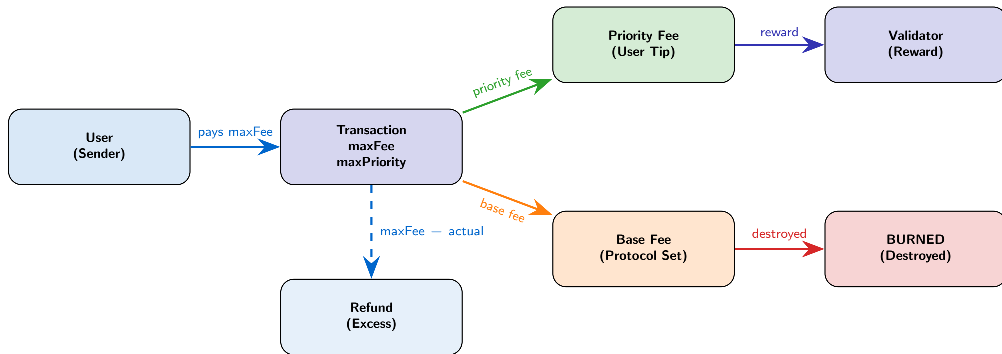
Operation	Max/Block
ETH transfers	∼1,428
ERC-20 transfers	∼460
Uniswap swaps	∼200
NFT mints	∼120
Contract deploys	∼5–15

Elasticity

The 2× buffer absorbs demand spikes while the fee mechanism ensures long-run convergence to 50% utilization.

Block

EIP-1559 Deep Dive: Fee Flow



Actual Fee Calculation

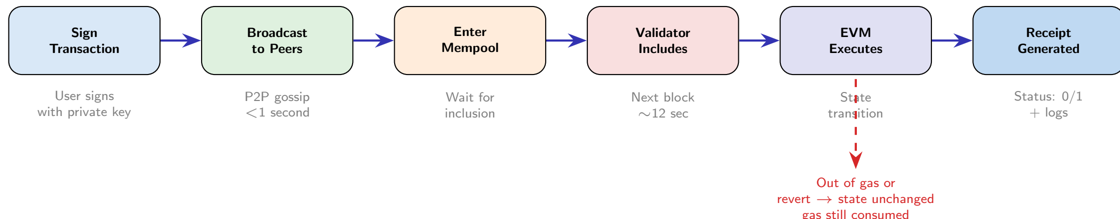
$$\text{effectiveGasPrice} = \min\big(\texttt{maxFee}, \texttt{baseFee} + \texttt{maxPriority}\big)$$

$$\text{totalFee} = \text{gasUsed} \times \text{effectiveGasPrice}$$

Worked Example

`maxFeePerGas`	50 Gwei
`maxPriorityFee`	2 Gwei
Current `baseFee`	30 Gwei
`gasUsed`	21,000
Effective price	$\min(50, 30 + 2) = 32$ Gwei
Total fee	$21{,}000 \times 32 = 672{,}000$ Gwei

Transaction Lifecycle



Mempool Dynamics

- Transactions ordered by `effectiveGasPrice` (highest first)
- Same-sender txs ordered by `nonce` (sequential)
- Stuck tx: low fee $\rightarrow$ replace with higher-fee tx (same nonce)
- Mempool is **not** consensus-critical – each node has its own

Finality Timeline

- **Included**: 12 seconds (1 slot)
- **Safe**: $\sim$6.4 minutes (1 epoch, 32 slots)
- **Finalized**: $\sim$13 minutes (2 epochs)
- After finalization: reverting requires $\geq$1/3 of all staked ETH to be slashed

Ether processes $\sim$1M transactions/day across 7,200 blocks (12-second slots)

Transaction Receipt & Logs

Transaction Receipt

- `status`: 1 (success) or 0 (fail)
- `transactionHash`: 32-byte id
- `blockNumber`: inclusion height
- `gasUsed`: actual gas consumed
- `cumulativeGasUsed`: block total
- `logs[]`: emitted events
- `logsBloom`: 256-byte filter

Log Entry Structure

- `address`: emitting contract
- `topics[0]`: event signature hash
- `topics[1]`: indexed param 1
- `topics[2]`: indexed param 2
- `data`: non-indexed params (ABI)

- Max 4 topics (incl. signature)
- Each topic = 32 bytes, filterable
- Data field: ABI-encoded, not indexed
- Logs cost 375 + 375/topic + 8/byte

Why Events Matter

Events are 5–100× cheaper than storage for recording historical data. DApps use `eth_getLogs` to reconstruct state from event history. The Bloom filter in each block header enables efficient log scanning.

Topic

= keccak256("Transfer(address,address,uint256)") for ERC-20 Transfer events

Estimating Gas Costs: Worked Example

Cost Formula

$$\text{Cost (ETH)} = \text{gasUsed} \times \text{effectiveGasPrice}$$

$$\text{Cost (USD)} = \text{Cost (ETH)} \times \text{ETH/USD price}$$

Unit Conversions

From		To
1 Gwei	=	10^9 Wei
1 ETH	=	10^9 Gwei
1 ETH	=	10^{18} Wei

Worked Example: ETH Transfer

Gas used	21,000
Gas price	30 Gwei
Cost (Wei)	$21{,}000 \times 30 \times 10^9$
Cost (ETH)	0.00063 ETH
ETH price	$3,000
Cost (USD)	**$1.89**

Common Operation Costs (at 30 Gwei)

Operation	Gas	USD
ETH transfer	21,000	$1.89
ERC-20 transfer	65,000	$5.85
ERC-20 approve	46,000	$4.14
Uniswap V3 swap	150,000	$13.50
NFT mint (ERC-721)	120,000	$10.80
Contract deploy	500,000+	$45+
Aave borrow	350,000	$31.50

Gas Price History

- 2021 bull market: 100–500 Gwei (NFT minting wars)
- 2022–23 bear market: 5–20 Gwei
- 2024 with L2 adoption: 10–40 Gwei typical
- During MEV events: spikes to 1000+ Gwei

Pro

Gas Optimization Patterns

1. Pack Storage Variables

Group uint128 + uint128 into one 256-bit slot instead of two slots.
Saves: ∼15,000 gas per slot

2. Calldata vs Memory

Use `calldata` for read-only function params (external functions).
Saves: ∼600 gas per parameter

3. Short-Circuit Logic

Place cheapest conditions first in `require` chains and `if` statements.
Saves: variable (avoids SLOAD)

4. Batch Operations

Combine multiple transfers in one tx. Amortize 21,000 base cost.
Saves: 21,000 per batched tx

5. Events vs Storage

Use `emit` for historical data that does not need on-chain reads.
Saves: 15,000–19,600 gas

6. Minimal Proxy (EIP-1167)

Clone contracts via 45-byte proxy instead of full deploy.
Saves: ∼90% deploy cost

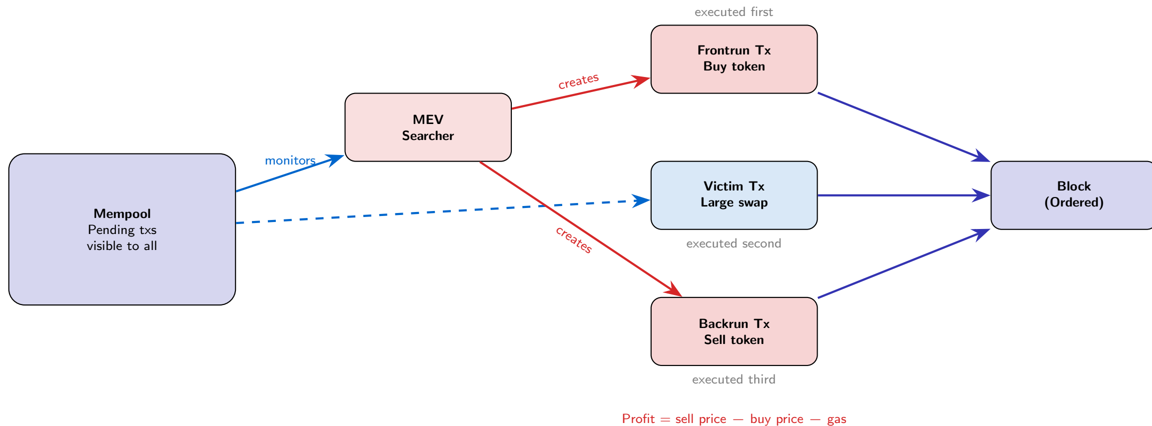
Rule of Thumb

Optimize storage first (SSTORE/SLOAD dominate costs), then reduce calldata size, then optimize computation. A single unnecessary `SSTORE` to a new slot costs more than 6,000 arithmetic operations.

Adva

Solidity 0.8.19+ supports custom errors (`error InsufficientBalance()`) saving ∼3,000 gas vs string reverts

MEV: Maximal Extractable Value



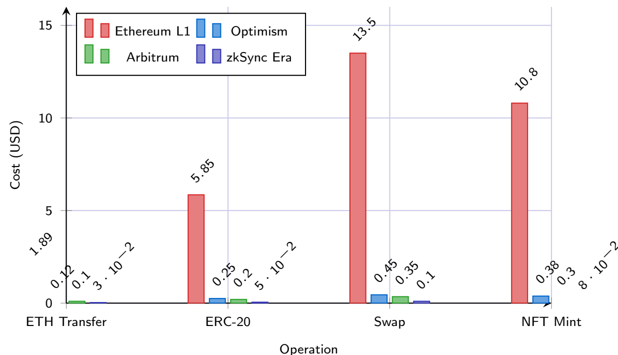
MEV Categories

- **Sandwich attacks**: front + backrun swaps
- **Arbitrage**: price differences across DEXes

MEV Mitigation

- **Flashbots Protect**: private tx submission
- **MEV Boost**: separate block building from proposing

Layer 2 Gas Savings



L2 Scaling Approaches

- **Optimistic Rollups** (Optimism, Arbitrum):
  - Execute off-chain, post data to L1
  - 7-day fraud proof window
  - 10–50 $\times$ cheaper
- **ZK Rollups** (zkSync, Polygon zkEVM):
  - Generate validity proofs (zk-SNARKs)
  - Instant finality (once proof posted)
  - 50–100 $\times$ cheaper

EIP-4844 Impact (Dencun)

- Introduces "blob" transactions for rollup data
- Separate fee market for blob space
- L2 costs dropped 10–100 $\times$ post-Dencun
- Target: $0.001 per L2 transaction

L2s collectively process ¿10 $\times$ more transactions than L1 – the rollup-centric roadmap is working

Gas & Transactions: Key Takeaways

Core Formulas

Transaction Cost:

$$\text{cost} = \text{gasUsed} \times \text{effectiveGasPrice}$$

Base Fee Adjustment:

$$b_{n+1} = b_n \times \left(1 + \frac{1}{8} \cdot \frac{g-t}{t}\right)$$

where $g = \text{gasUsed}$, $t = \text{target (15M)}$

Effective Gas Price:

$$p = \min(\texttt{maxFee},\ b + \texttt{tip})$$

Key Numbers to Remember

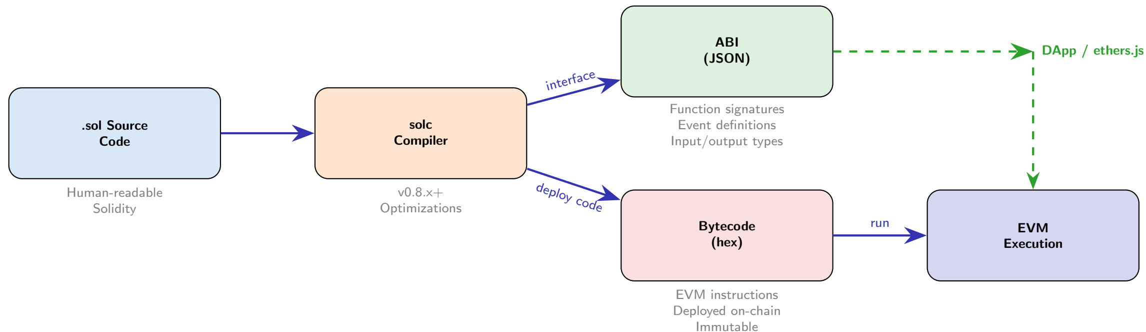
Item	Value
ETH transfer gas	21,000
Block gas limit	30,000,000
Block gas target	15,000,000
SSTORE (new slot)	20,000 gas
SLOAD (cold)	2,100 gas
ADD opcode	3 gas
Block time	12 seconds
Max fee change/block	$\pm 12.5\%$

Optimization Priority

1. Minimize storage writes (SSTORE)
2. Use events for historical data
3. Pack variables into 256-bit slots
4. Consider L2 for cost-sensitive applications

Gas is the bridge between economic incentives and computational resources – the heart of Ethereum's mechanism design

Solidity: The Smart Contract Language



Language Features

- Statically typed
- Inheritance (multiple)
- Modifiers (decorators)
- Events for logging
- Error handling (require/revert)

Alternatives

- **Vyper**: Python-like, safer defaults
- **Yul**: Low-level, assembly-like
- **Huff**: Ultra-low-level, max gas efficiency
- **Fe**: Rust-inspired, experimental

Key Stats

- ~90% of contracts use Solidity
- >300K verified contracts on Etherscan
- Latest stable: v0.8.28
- Built-in overflow checks (0.8+)

Contract Structure

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/access/Ownable.sol";

contract SimpleVault is Ownable {
  // State variables (storage)
  mapping(address => uint256) public balances;
  uint256 public totalDeposits;

  // Events
  event Deposit(address indexed user, uint256 amount);
  event Withdrawal(address indexed user, uint256 amount);

  // Constructor
  constructor() Ownable(msg.sender) {}
```

```
  // Modifier
  modifier hasBalance(uint256 amount) {
    require(balances[msg.sender] >= amount,
            "Insufficient balance");
    _;
  }

  // External function (payable)
  function deposit() external payable {
    balances[msg.sender] += msg.value;
    totalDeposits += msg.value;
    emit Deposit(msg.sender, msg.value);
  }

  // External function with modifier
  function withdraw(uint256 amount)
    external hasBalance(amount) {
    balances[msg.sender] -= amount;
    totalDeposits -= amount;
    payable(msg.sender).transfer(amount);
    emit Withdrawal(msg.sender, amount);
  }
}
```

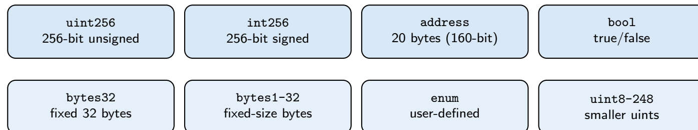
Anatomy Checklist

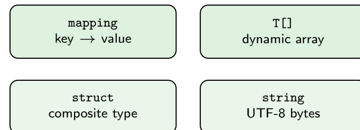
License (SPDX) → **Pragma** (compiler version) → **Imports** → **Contract declaration** (inheritance) → **State variables** → **Events** → **Constructor** → **Modifiers** → **Functions**

Conv

state variables first, then events, constructor, modifiers, external/public, internal/private functions

Data Types Overview





Gas Cost by Type

Type	Size	Storage Cost
uint256	32 bytes (1 slot)	20,000 gas
address	20 bytes	20,000 gas
bool	1 byte (1 slot!)	20,000 gas
uint128 + uint128	32 bytes (1 slot)	20,000 gas
mapping entry	32 bytes/value	20,000 gas

Important Notes

- A bool alone wastes 31 bytes of a 32-byte slot
- Pack smaller types together: uint128 + uint128 = 1 slot
- mapping keys are not iterable (no .length)
- string is dynamically sized – expensive for long strings
- address payable can receive ETH via .transfer()

EVM operates on 256-bit words – smaller types save gas only when packed together in the same storage slot

Storage vs Memory vs Calldata

Storage (Persistent)

- On-chain, survives between calls
- Key-value: 256-bit $\longrightarrow$ 256-bit
- SSTORE: 20,000 / 5,000 gas
- State variables live here

← permanent

Memory (Temporary)

- Wiped after each external call
- Byte-addressable, expandable
- MLOAD/MSTORE: 3 gas (+ expansion)
- Function-local variables, ABI decode

← tx lifetime

Calldata (Read-Only Input)

- Transaction input data
- Immutable during execution
- CALLDATALOAD: 3 gas
- Use for external function params

← read-only

Cost Comparison

Location	Read	Write
Storage (cold)	2,100	20,000
Storage (warm)	100	5,000
Memory	3	3
Calldata	3	–

When to Use Each

- **Storage**: state that must persist (balances, ownership)
- **Memory**: intermediate computations, return values, struct construction
- **Calldata**: external function parameters that are read but not modified

Optimization Rule

Cache storage reads in memory variables: read once from storage (2,100 gas), reuse from memory (3 gas).

Functions & Visibility

	State-Changing	`view` (reads only)	`pure` (no state)
`public`	✓ All callers	✓ Free from contract	✓ No gas (off-chain)
`external`	✓ External only	✓ Cheaper calldata	✓ External only
`internal`	✓ This + children	✓ This + children	✓ This + children
`private`	✓ This only	✓ This only	✓ This only

Visibility Rules

- **public**: callable by anyone; auto-generates getter for state vars
- **external**: only callable from outside (saves gas via `calldata`)
- **internal**: this contract + derived contracts (like `protected`)
- **private**: only this contract (not derived contracts)

State Mutability

- **(default)**: can read and write state, costs gas
- **view**: reads state but never modifies (free if called externally)
- **pure**: no state access at all (e.g., math utilities)
- **payable**: can receive ETH with the call

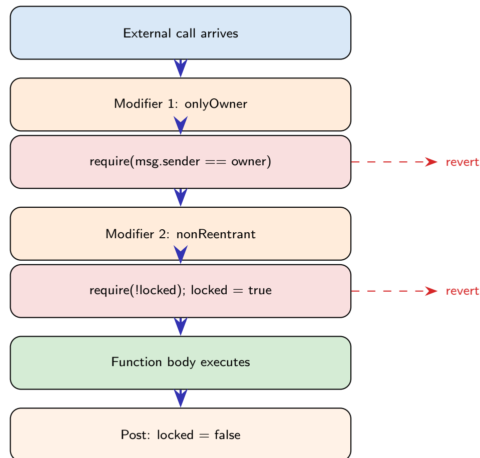
Gas Tip

Function Modifiers

```
// Access control modifier
modifier onlyOwner() {
  require(msg.sender == owner,
          "Not the owner");
  _;  // function body executes here
}

// Reentrancy guard modifier
modifier nonReentrant() {
  require(!locked, "Reentrant call");
  locked = true;
  _;  // function body
  locked = false;
}

// Usage: stacking modifiers
function withdraw(uint256 amount)
  external
  onlyOwner
  nonReentrant
{
  payable(msg.sender).transfer(amount);
}
```



Modifier Best Practices

Events & Logging

```
// Event declaration (max 3 indexed)
event Transfer(
  address indexed from,
  address indexed to,
  uint256 value
);

event Approval(
  address indexed owner,
  address indexed spender,
  uint256 value
);

// Emitting events
function transfer(address to, uint256 amount)
  external returns (bool) {
  balances[msg.sender] -= amount;
  balances[to] += amount;
  emit Transfer(msg.sender, to, amount);
  return true;
}
```

Log Entry in Receipt

Gas Cost Breakdown

- Base log cost: 375 gas
- Per topic: 375 gas
- Per data byte: 8 gas
- Transfer event total: ∼1,500 gas
- vs SSTORE: 20,000 gas
- **Savings: 12× cheaper!**

When to Use Events vs Storage

Use **events** for data that only needs off-chain access (audit trails, UI updates, historical queries). Use **storage** only for data that contracts need to read on-chain. Events are not accessible from within smart contracts.

Mappings & Arrays: Storage Layout

Contract Storage Layout

Slot 0: `uint256 totalSupply`
Slot 1: `mapping balances` (empty marker)
Slot 2: `uint256[] ids` (array length)

Mapping: balances[addr]

location = keccak256(addr || 1)

key concatenated with slot number, then hashed to find storage position

Dynamic Array: ids[i]

location = keccak256(2) + i

slot number hashed for base, index added for element offset

Mapping Properties

- No iteration (no `.length`, no `.keys()`)
- All keys implicitly exist (default: 0/false)
- Cannot be returned from functions
- Nested: `mapping(a => mapping(b => v))`
- Storage position: deterministic via keccak256

Dynamic Array Properties

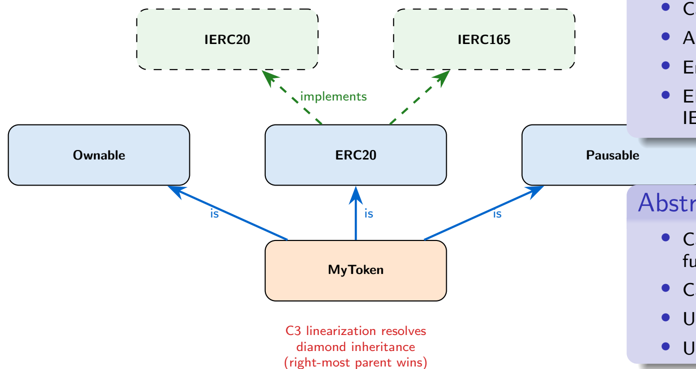
- `.push()`, `.pop()`, `.length`
- Elements stored contiguously from keccak256(slot)
- Iteration possible but gas scales linearly
- Deletion leaves gaps (use swap-and-pop pattern)

Design Pattern

Combine mapping + array for iterable mappings: `mapping(addr => uint)` for O(1) lookup + `address[]` for enumeration.

Stora

Inheritance & Interfaces



Interfaces

- Define function signatures without implementation
- Cannot have state variables or constructors
- All functions must be `external`
- Enable polymorphism and loose coupling
- ERC standards defined as interfaces (IERC20, IERC721)

Abstract Contracts

- Can have both implemented and unimplemented functions
- Cannot be deployed directly
- Use `virtual` for overridable functions
- Use `override` in derived contracts

C3 Linearization

For `contract MyToken is Ownable, ERC20, Pausable`: resolution

Key ERC Interfaces

Standard	Purpose

Error Handling

	require(cond, msg)	assert(cond)	revert CustomError()
Purpose	Input validation, access control, preconditions	Invariant checks, internal bugs	Custom errors with parameters (0.8.4+)
Gas on failure	Refunds remaining gas	Consumes **all** remaining gas (pre-0.8.0); refunds (0.8.0+)	Refunds remaining gas
Use when	External input might be invalid	Condition should *never* be false	Need typed errors with data
Error data	String message (expensive)	Panic(uint256) error code	4-byte selector + ABI params
Gas cost	∼2,500 + string storage	∼2,500 (0.8+)	∼150 (just selector!)
Example	`require(bal >= amt, "Low")`	`assert(supply == sum)`	`revert InsufficientBalance()`

Custom Errors (Recommended)

- Defined at contract level: `error InsufficientBalance(uint256 available, uint256 required)`
- Save ∼90% gas vs string revert messages
- ABI-encodable, can carry context data
- Supported by all modern tooling (ethers.js, Foundry)

try/catch (External Calls)

- Catches reverts from **external** calls only
- Cannot catch errors in the same contract
- Catch clauses: `Error(string)`, `Panic(uint256)`, generic `(bytes)`
- Essential for composability with untrusted contracts

Migration Path

Replace `require(cond, "string")` with custom errors for 90% gas savings on reverts. Solidity 0.8.26+ supports `require(cond,`

Common Design Patterns

Ownable (Access Control)

Single owner with transfer capability.
Use: admin functions, upgrades.
See also: AccessControl (role-based).

Pausable (Emergency Stop)

Owner can pause/unpause contract.
Use: circuit breaker during exploits.
Pattern: `whenNotPaused` modifier.

Proxy (Upgradeability)

Storage in proxy, logic in implementation.
DELEGATECALL forwards execution.
Standards: EIP-1967, UUPS, Beacon.

Factory (Create Contracts)

Deploy new contracts from a template.
Use: token launches, pool creation.
Pattern: CREATE2 for deterministic addr.

State Machine (Workflow)

Enum-based state transitions.
Use: crowdsales, governance, escrow.
Modifier enforces valid transitions.

Pull Payment (Safe Transfer)

Recipients withdraw instead of push.
Use: avoid reentrancy on sends.
Pattern: credit balance, let user claim.

OpenZeppelin Contracts

- Industry-standard library of audited patterns
- Ownable, AccessControl, Pausable, ReentrancyGuard
- ERC20, ERC721, ERC1155 implementations
- Governor (on-chain governance), TimelockController

Pattern Selection Guide

Need admin control?	Ownable / AccessControl
Need upgrades?	UUPS Proxy
Need emergency stop?	Pausable
Deploying many copies?	Factory + EIP-1167
Complex workflow?	State Machine

“Don’t roll your own crypto, don’t roll your own access control” – always use audited libraries like OpenZeppelin

Solidity Basics: Quick Reference

Type System

Value types: `uint256`, `int256`, `address`, `bool`, `bytes32`
Reference types: `mapping`, `array[]`, `struct`, `string`
Locations: `storage` (persistent) — `memory` (temp) — `calldata` (read-only)

Functions

Visibility: `public` — `external` — `internal` — `private`
Mutability: (default) — `view` — `pure` — `payable`
Modifiers: `onlyOwner`, `nonReentrant`, custom
Tip: prefer `external` + `calldata` for gas savings

Error Handling

`require(cond, msg)` – input validation
`revert CustomError()` – gas-efficient (0.8.4+)
`assert(cond)` – invariant checks

Events

```
event Transfer(address indexed from, address indexed to, uint256 value)
```

Max 3 indexed params (topics), rest in data
12× cheaper than storage for historical records

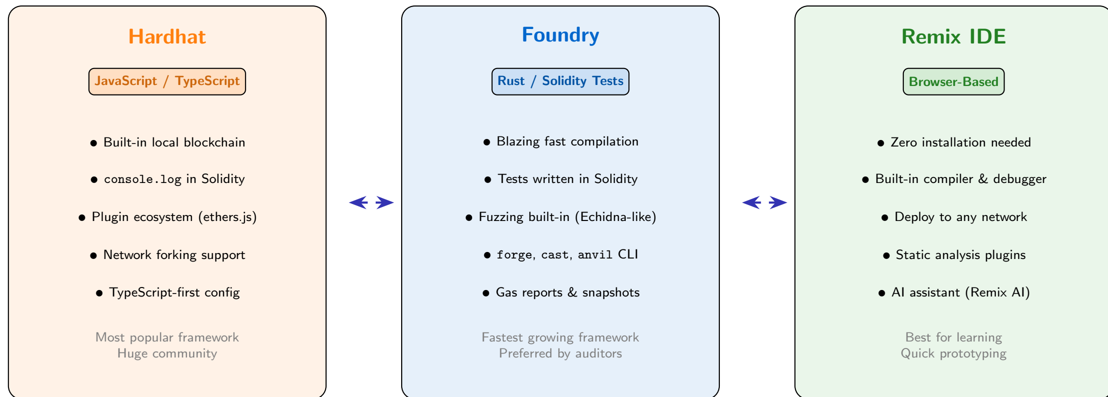
Storage Layout

Each slot = 256 bits = 32 bytes
Mapping: `keccak256(key || slot)`
Dynamic array: `keccak256(slot) + index`
Pack small types: `uint128 + uint128` = 1 slot

Design Patterns

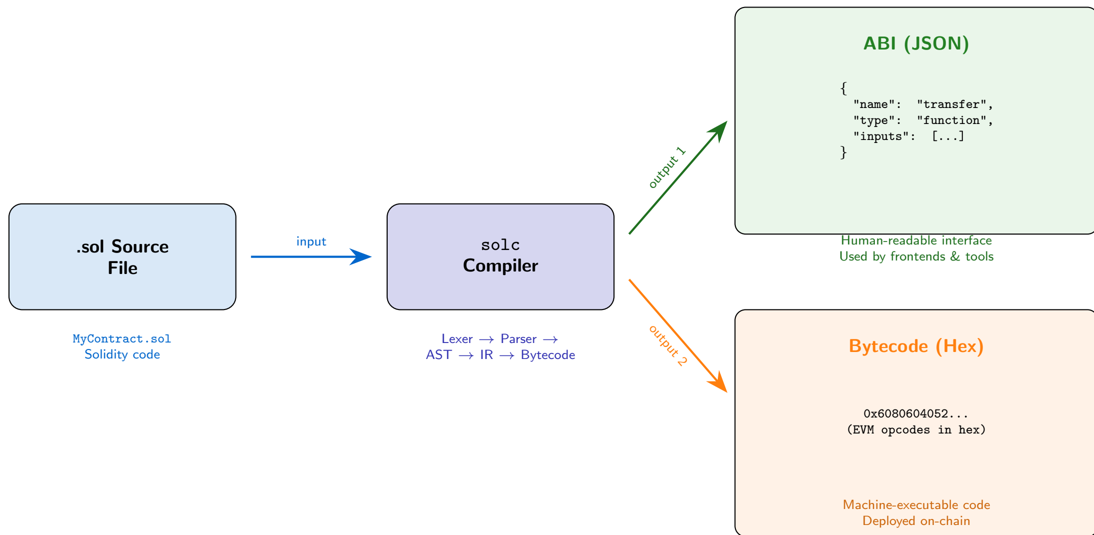
Ownable (access) — Pausable (emergency)
Proxy (upgrades) — Factory (deploy)
State Machine (workflow) — Pull Payment (safety)
Always use OpenZeppelin for standard patterns

Development Environment

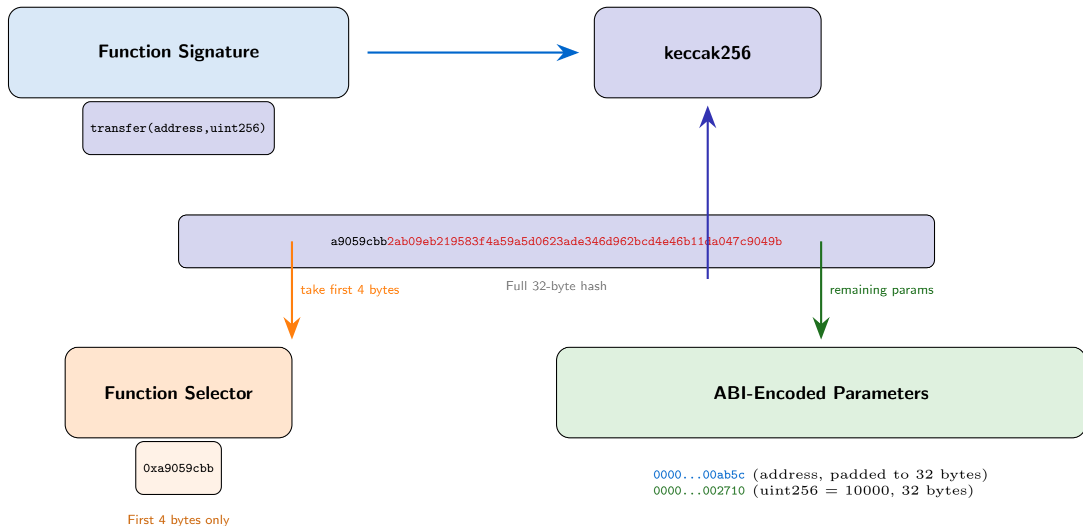


Start with Remix for learning, graduate to Hardhat/Foundry for production – many teams use Foundry for tests + Hardhat for scripts

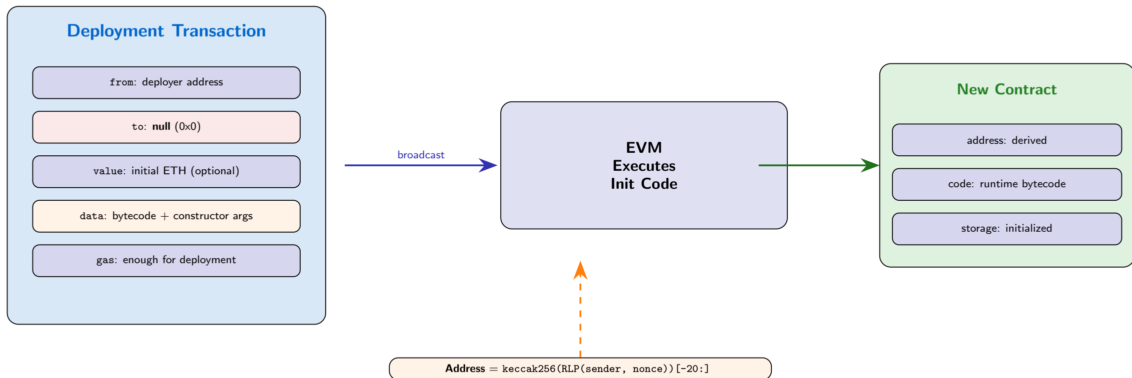
Compilation Process



ABI: The Contract Interface



Deployment Transaction



Key Insight

The `data` field contains *init code* (constructor + deployment logic) that runs once, returning the *runtime bytecode* that is permanently stored on-chain. The constructor itself is **not** part of the deployed code.

Contr

CREATE vs CREATE2

CREATE (Opcode)

```
addr = keccak256(sender, nonce)
```

- Address depends on sender's nonce
- Nonce increments each deploy
- **Non-deterministic** across chains
- Cannot predict address before deploy
- Default behavior since genesis

Use case: standard deployments where address doesn't matter

CREATE2 (EIP-1014)

```
addr = keccak256(0xff, sender,
salt, keccak256(bytecode))
```

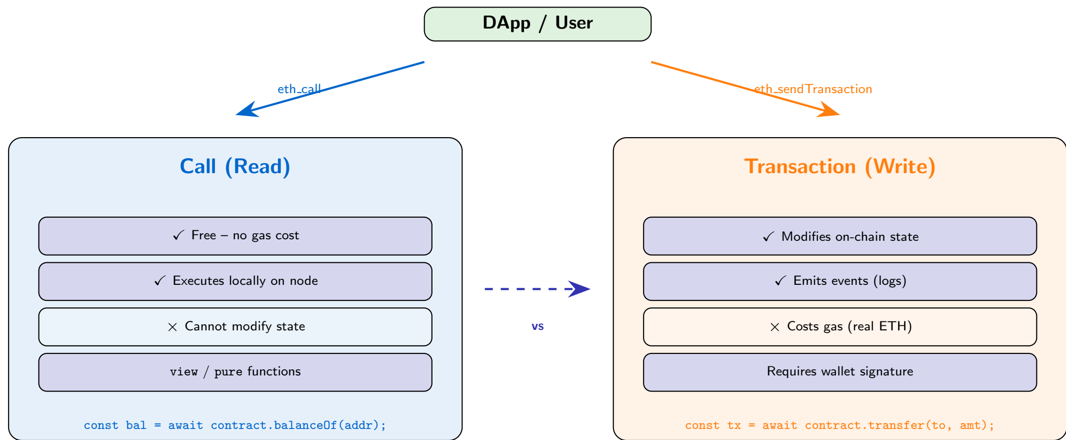
- Address is **deterministic**
- Same inputs $\longrightarrow$ same address always
- Works across chains (same sender)
- Salt = user-chosen 32 bytes

Use cases: counterfactual addresses, cross-chain deploys, factory patterns

Why CREATE2 Matters

Deterministic addresses enable **counterfactual instantiation**: you can compute a contract's address *before* deploying it, allowing users to send funds to a contract that doesn't exist yet (used in state channels, account abstraction, and cross-chain protocols).

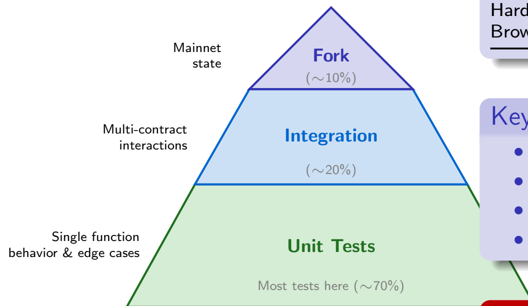
CREA



Common Gotcha

Calling a state-changing function with `eth_call` will *simulate* it (returning a result) but **not** persist changes. You must send a **transaction** for state to update on-chain. Conversely, calling a `view` function via transaction wastes gas.

Testing Smart Contracts



Testing Frameworks

Tool	Language	Speed
Foundry	Solidity	Very fast
Hardhat	JS/TS	Moderate
Brownie	Python	Moderate

Key Testing Techniques

- **Fuzzing**: random inputs find edge cases
- **Invariant testing**: properties that must always hold
- **Fork testing**: replay against real mainnet state
- **Gas snapshots**: detect regressions

Coverage Target

Aim for >95% line coverage. Critical paths (fund transfers, access control) must have 100% branch coverage.

"Code without tests is broken by design" – a $100M contract with 50% coverage is an audit red flag

Security Vulnerabilities

Reentrancy

CRITICAL

Attacker re-enters function before state update. The DAO lost $60M.

`Fix: checks-effects-interactions`

Integer Overflow

HIGH

Arithmetic wraps around max value. Fixed in Solidity 0.8+ with built-in checks.

Fix: use Solidity $\geq$ 0.8.0

Access Control

HIGH

Missing or weak auth lets anyone call admin functions. Parity wallet: $150M frozen.

`Fix: OpenZeppelin Ownable/Roles`

Front-Running

MEDIUM

Mempool observers sandwich user txs for profit. MEV bots extract $1M+/day.

`Fix: commit-reveal, Flashbots`

Oracle Manipulation

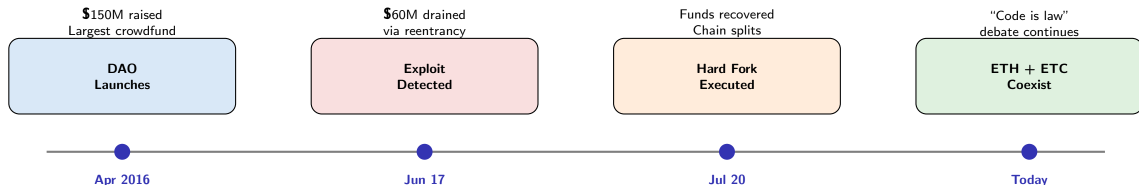
MEDIUM

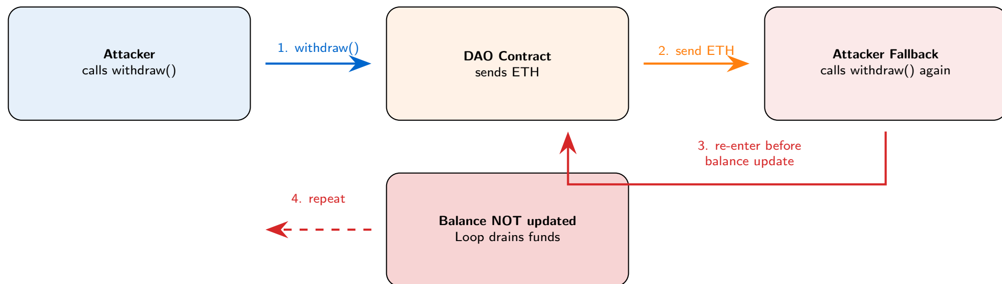
Flash loan manipulates price oracle in single tx. $100M+ stolen across DeFi.

`Fix: TWAP, Chainlink oracles`

Smart contract vulnerabilities have caused >$5B in losses – security is not optional, it is existential

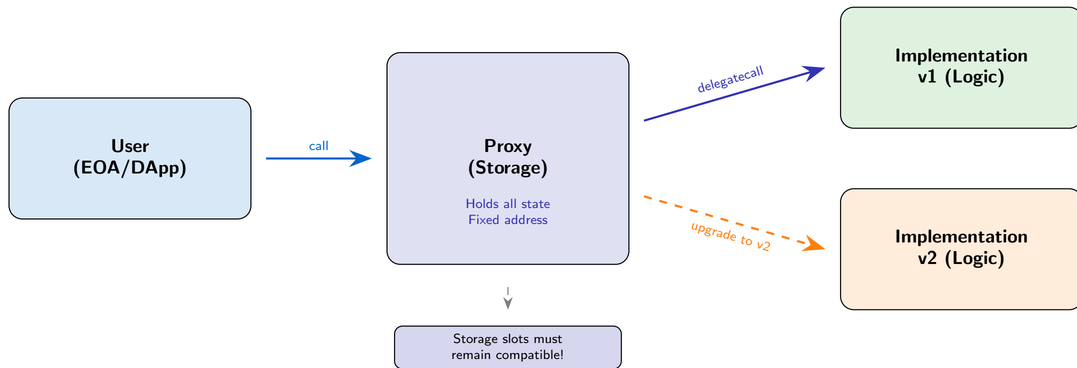
The DAO Hack Case Study





The Fix: Checks-Effects-Interactions Pattern

Upgradeability Patterns

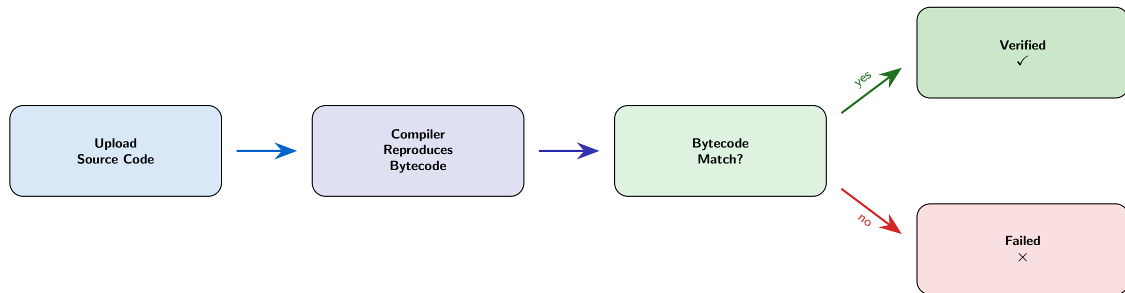


Transparent Proxy (TPP)

- Admin calls → proxy logic (upgrade)
- User calls → delegatecall to implementation
- Clear separation of concerns
- Higher gas: admin check on every call

UUPS (EIP-1822)

- Upgrade logic lives in *implementation*
- Proxy is minimal (cheaper deploys)
- Implementation must include upgrade function
- Risk: forgetting upgrade function = bricked



Verified Contract

✓ **Green checkmark on Etherscan**

- Source code readable by anyone
- Read/Write contract UI available
- Function names visible in tx decoder
- Users can audit the logic

Unverified Contract

× **No source code visible**

- Only raw hex bytecode shown
- Function calls show as hex selectors
- Cannot easily audit logic
- **Major trust red flag**

Verification Methods

Deploy & Interact: Section Summary

Deployment Checklist

- ✓ **Compile**: `solc` or framework CLI
- ✓ **Test**: unit, integration, fork, fuzz
- ✓ **Audit**: internal review + external firm
- ✓ **Deploy**: testnet first, then mainnet
- ✓ **Verify**: source code on Etherscan
- ✓ **Monitor**: events, alerts, dashboards

Key Tools

Frameworks: Hardhat, Foundry, Remix
Testing: Forge, Echidna, Mythril
Deploy: Hardhat Ignition, forge create
Monitor: Tenderly, OpenZeppelin Defender
Verify: Etherscan, Sourcify

Security Essentials

Top vulnerabilities:

- Reentrancy (checks-effects-interactions)
- Access control (role-based auth)
- Oracle manipulation (TWAP + Chainlink)
- Front-running (commit-reveal, Flashbots)

Patterns that protect:

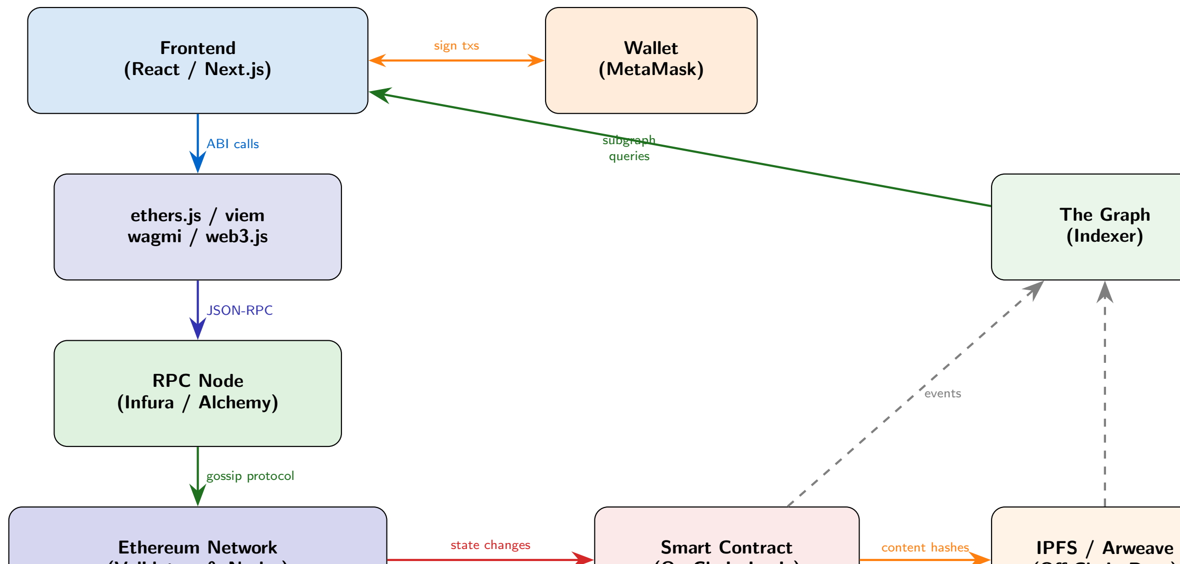
- OpenZeppelin libraries
- Proxy patterns for upgradeability
- Timelocks for governance
- Circuit breakers (Pausable)

Golden Rule

Immutable code + real money = **zero room for error**. Every deployed contract should be tested, audited, and verified before holding user funds.

Next section: Summary, token standards, DeFi building blocks, and the full-stack DApp architecture

Full Stack DApp Architecture



Token Standards Overview

ERC-20: Fungible Tokens

Currencies, Stablecoins, Governance

`transfer(to, amount)`

`approve(spender, amount)`

`transferFrom(from, to, amount)`

ERC-721: NFTs

Art, Gaming, Identity

`ownerOf(tokenId)`

`safeTransferFrom(from, to, tokenId)`

`tokenURI(tokenId)`

ERC-1155: Multi-Token

Gaming Items, Mixed Collections

`balanceOfBatch(owners, ids)`

`safeTransferFrom(from, to, id, amt)`

`Fungible + NFT in one contract`

ERC-4626: Tokenized Vaults

Yield, Lending, Staking

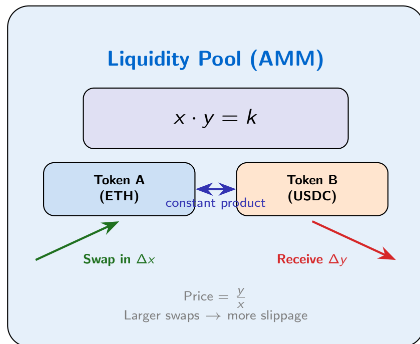
`deposit(assets, receiver)`

`withdraw(assets, receiver, owner)`

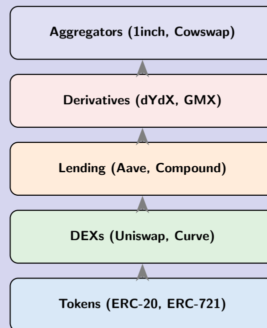
`convertToShares(assets)`

Token standards are interfaces (ERCs) – they define what functions exist, not how they work internally

DeFi Building Blocks



Money Legos: Composability



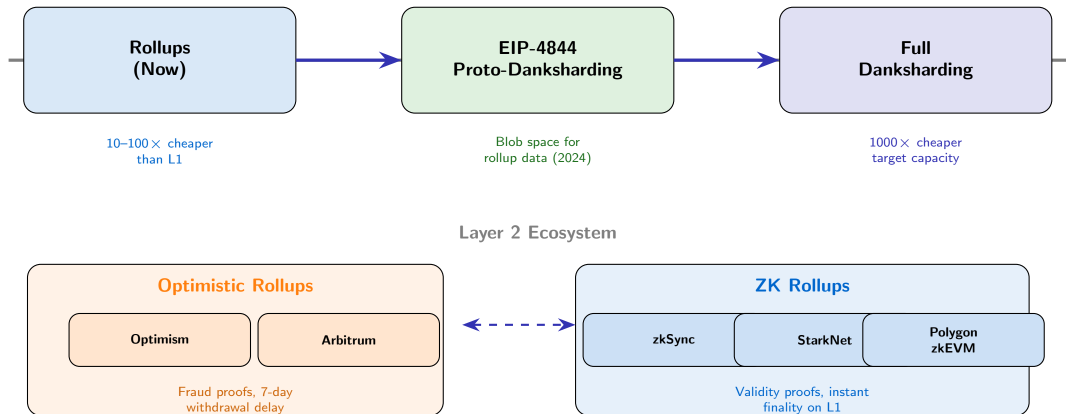
Flash Loans

Borrow *any amount* with zero collateral – must repay in the **same transaction**. Enables arbitrage, liquidations, and collateral swaps atomically.

DeFi'

composability means protocols can be combined permissionlessly – a single tx can touch 5+ protocols

Ethereum Scaling Roadmap

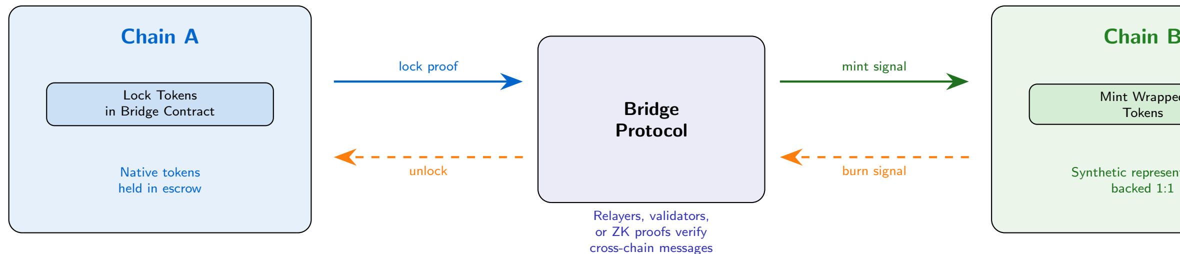


The Rollup-Centric Roadmap

Ethereum L1 focuses on **security and data availability**. Execution moves to L2 rollups that inherit L1 security. EIP-4844 introduced "blobs" – dedicated cheap data space for rollups, reducing L2 fees by 10–100×.

As of 2024, Ethereum L2s process more transactions than L1 – the rollup-centric roadmap is working

Cross-Chain & Bridges



Bridge Trust Models

- **Trusted**: multisig validators (fast, centralized)
- **Optimistic**: fraud proofs (slow, trust-minimized)
- **ZK**: validity proofs (fast, trust-minimized)
- **Native**: protocol-level (most secure)

Bridge Security

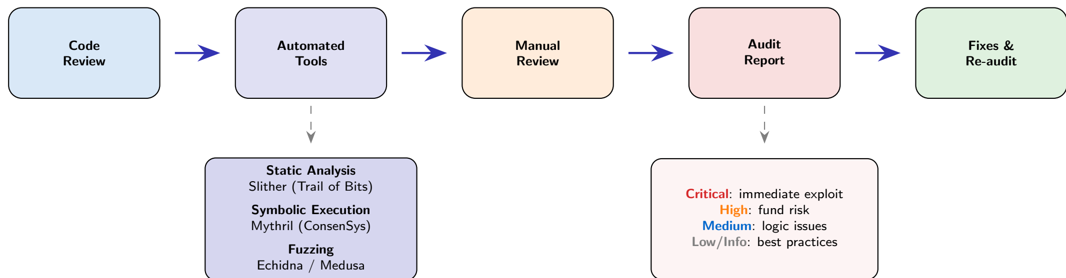
Bridges are high-value targets – over **$2B+ stolen** from bridge hacks:

- Ronin Bridge: $625M (compromised keys)
- Wormhole: $320M (signature bypass)
- Nomad: $190M (verification bug)

"Brid

are the weakest link in crypto security" – cross-chain value transfer remains an open research problem

Smart Contract Auditing



Audit Economics

Scope	Typical Cost
Simple token	\$5K–\$15K
DeFi protocol	\$50K–\$200K
Complex protocol	\$200K–\$500K

Timeline: 1–8 weeks depending on complexity

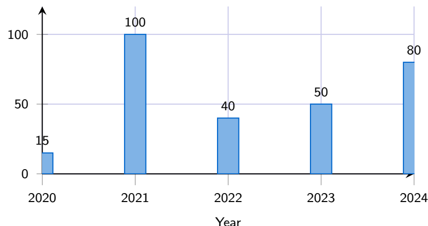
Audits Are Necessary But Not Sufficient

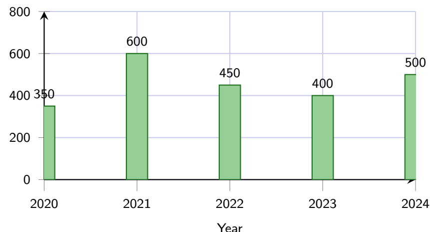
An audit is a *point-in-time* review. It does not guarantee zero bugs. Combine with:

- Bug bounty programs (Immunefi)
- Formal verification
- Continuous monitoring
- Upgrade timelocks

Top audit firms: Trail of Bits, OpenZeppelin, Consensys Diligence, Spearbit, Cyfrin – demand exceeds supply

Ethereum by the Numbers





ETH Burned Since EIP-1559

- Over **4M+ ETH** burned (>$8B+ at peak)
- Net deflationary during high activity
- "Ultrasound money" narrative
- Burn rate: 1–5 ETH/min average

Validator Ecosystem

- **900K+** active validators
- 32 ETH minimum stake
- ∼28M ETH staked (∼23% supply)
- Lido, Coinbase, Rocket Pool dominate

Ether processes ∼1M transactions/day on L1, with L2s adding another ∼5M+/day – the ecosystem continues to grow

Career Paths in Ethereum

Smart Contract Developer

$100K – $200K

Solidity, Foundry, OpenZeppelin, testing, gas optimization, auditing basics

Security Auditor

$150K – $300K

EVM internals, formal verification, Slither/Mythril, CTF experience

MEV Researcher

$200K+

Game theory, Flashbots, mempool analysis, Rust, low-latency systems

Protocol Engineer

$150K – $250K

Rust/Go, consensus algorithms, P2P networking, cryptography

DApp Developer

$80K – $150K

React, ethers.js/viem, wagmi, The Graph, IPFS, UI/UX

Getting Started

Path 1: CryptoZombies → Foundry → CTFs (Ethernaut, Damn Vulnerable DeFi) → build projects → contribute to protocols. **Path 2**: Web2 dev → learn ethers.js → build DApps → learn Solidity → full-stack Web3.

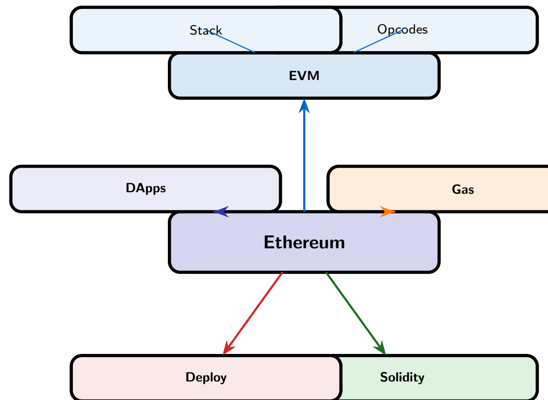
Course Takeaways

Five Key Principles

- ✓ **Ethereum = programmable blockchain** with a Turing-complete EVM enabling arbitrary on-chain logic
- ✓ **Gas prevents abuse** and pays validators for computation – EIP-1559 makes fees predictable
- ✓ **Solidity enables complex on-chain logic** with types, storage, events, and design patterns
- ✓ **Security is paramount** – audit everything, use established patterns, test exhaustively
- ✓ **The ecosystem is composable** (money legos) – protocols build on protocols permissionlessly

Remember

Smart contracts are **immutable** and handle **real value**. Every line of code is a potential attack surface. Think like an attacker, build like a defender.



Next lecture: Build your own ERC-20 token!