#7 Gas, Testing, and Tooling

Lecture Notes for CS190N: Blockchain Technologies and Security October 20, 2025

In this lecture, we learn how to write contracts that are efficient, testable, and professionally engineered. We begin by turning gas into a working cost model, covering transaction pricing under EIP-1559, the price of storage reads and writes, cold and warm access under EIP-2929, memory expansion, and how compiler optimizer settings influence both deployment and execution. We then translate the model into practice with micro techniques that reduce writes, pack slots, favor calldata, cache storage reads, and apply unchecked arithmetic only when a proof of safety exists, and we measure every change with traces, gas reports, and snapshots. Next, we build assurance through unit tests that check state, reverts, and events, followed by property based tests that state invariants and use fuzzing to search for counterexamples. Finally, we assemble a toolchain that turns these habits into a repeatable workflow, using Foundry or Hardhat together with Slither, Echidna, Mythril, and Tenderly for analysis, testing, and debugging. By the end, you will be able to predict where gas is spent, implement and verify optimizations that matter, and maintain a codebase whose behavior is demonstrated by tests rather than assumed.

1 INTRODUCTION

Smart contracts run in an adversarial and resource constrained environment. Deployed bytecode is immutable, state changes are costly, and bugs can have direct financial impact. This note focuses on three pillars that determine real world viability: gas efficiency, systematic testing, and a professional toolchain. We first build a concise cost model for the EVM and show how it shapes design choices. We then present unit and property testing patterns that surface bugs before deployment. Finally, we outline the tools that make these practices repeatable at scale so that the same quality bar can be enforced across contributors and over time.

2 GAS OPTIMIZATION PRINCIPLES: THE EVM'S ECONOMIC MODEL

Efficient code follows the cost model. The EVM prices work across opcodes, memory growth, and persistent storage. Understanding these prices is useful because it explains why some refactorings matter and others do not. This section states what is needed to predict costs and to choose cheaper designs, then anchors each idea with a minimal example that you can build on in your own experiments.

2.1 The Gas Model: Execution Limits and Fees

Gas bounds execution and prices it. A transaction specifies `gasLimit` and pays `gasUsed` × `effectiveGasPrice`. Under EIP 1559, `effectiveGasPrice` equals `baseFee + priorityFee`. Intrinsic cost is 21,000 plus calldata bytes, where zero bytes cost 4 gas and non zero bytes cost 16 gas. These components interact in predictable ways, which means you can estimate cost before you write code and then validate the estimate with a trace.

Example (Equation 1). A call that uses 100,000 gas at `baseFee` = 20 gwei and `priorityFee` = 2 gwei pays

$$\text{fee} = 100{,}000 \times (20 + 2)\ \text{gwei}. \tag{1}$$

This small calculation is enough to compare two code paths when their gas differs by only a few thousand units and to decide whether a micro optimization is worth the added complexity.

2.2 Storage Costs and Refunds

Persistent state dominates cost. `SSTORE` pricing depends on the before and after values and on whether the slot was cold or warm in the current transaction. A compact summary appears in

Table 1. Refunds for clearing state are credited only after successful completion and are capped post London, as noted under Table 1. In practice this means designs that batch writes and avoid churn are cheaper and more predictable to operate.

Example (Listing 1). First touch a slot, then touch it again within the same transaction, and observe that the cold surcharge disappears on the second access. This pattern explains why grouping updates by key reduces cost even if the number of logical assignments stays the same.

Listing 1. Cold versus warm storage touch in one transaction

```
contract ColdWarm {
    uint256 public x;
    function touchTwice(uint256 a, uint256 b) external {
        x = a; // first write, slot is cold, more expensive
        x = b; // second write, slot is warm, cheaper
    }
}
```

Table 1. SSTORE pricing with cold surcharge and refunds

State Transition	Base SSTORE	Cold SLOAD	Net (Cold)	Refund
Zero → Non Zero	20,000	2,100	22,100	0
Non Zero → Zero	5,000	2,100	7,100	15,000
Non Zero → Non Zero (diff)	5,000	2,100	7,100	0
Non Zero → Non Zero (same)	100	2,100	2,200	0

Note. The *Net (Cold)* column assumes the first access in the transaction. Warm repeats do not pay 2,100 again. Refunds are credited only after success, and total refunds are capped per transaction.

2.3 Cold and Warm Access

With EIP 2929, the first access to an account or slot in a transaction is cold. Later accesses are warm and cheaper. Since SSTORE reads before it writes, the cold surcharge influences writes as summarized in Table 1. This favors code that stages work on the stack and in memory, then commits a single write per slot, rather than interleaving reads and writes across many slots.

2.4 Memory Expansion

Memory is transient and priced by the highest offset touched. Expansion cost grows with size, so staging near offset 0 is cheaper. The model is simple enough to guide layout decisions, for example placing return buffers at low addresses and avoiding sparse writes that jump to high offsets early in execution.

Example (Listing 2). Returning 32 bytes from offset 0 is cheaper than from a high offset. This is a minimal demonstration of why many standard libraries place return data at zero unless there is a strong reason to reserve space.

Listing 2. Memory return at low and high offsets

```
contract MemDemo {
    function low() external pure returns (bytes32 r) {
        assembly { mstore(0x00, 0x42) r := mload(0x00) } // small expansion
    }
```

```
    function high() external pure returns (bytes32 r) {
        assembly { mstore(0x4000, 0x42) r := mload(0x4000) } // large expansion
    }
}
```

2.5 Compiler Optimizer and runs

The optimizer trades deployment size for execution cost. Low `runs` prefers smaller bytecode. High `runs` favors cheaper calls. Choose based on expected call volume and on whether the contract is long lived. Frozen values avoid storage entirely and keep access as a simple bytecode constant. A quick comparison appears in Table 2 and the minimal pattern in Listing 3.

Example (Listing 3). Both variables avoid `SLOAD` at runtime, and neither requires an `SSTORE` during deployment. This reduces one time and recurring costs without changing behavior.

Listing 3. Avoid storage with `constant` and `immutable`

```
contract Frozen {
    uint256 public constant DECIMALS = 18; // embedded at compile time
    address public immutable owner; // set once in constructor
    constructor() { owner = msg.sender; } // no storage writes afterward
}
```

Table 2. `constant` versus `immutable`

Aspect	constant	immutable
Assignment time	Compile time	Constructor time
Gas mechanism	Value embedded in bytecode	Value embedded in bytecode
Typical use	`uint256 constant DECIMALS=18;`	`address immutable owner=msg.sender;`

3 MICRO LEVEL TECHNIQUES FOR GAS OPTIMIZATION

The goal is to move work to cheaper locations and to reduce expensive operations. Each subsection introduces one design rule and points to a minimal example or table. When you apply these rules, measure their effect with a gas report so that changes stay justified by data.

3.1 Slot Packing

Pack smaller fields into one 256 bit slot to reduce writes. Order fields from wide to narrow, or group narrow fields together. The benefit comes from performing fewer `SSTORE` operations on write heavy paths, even though some accesses may require masking and shifting.

Example (Listing 4). The packed layout performs fewer `SSTORE` writes on updates, which is visible in a gas report during tests that set both fields together.

Listing 4. Pack fields to reduce storage writes

```
contract Packing {
    struct Unpacked { uint64 a; uint256 b; uint64 c; } // 3 slots
    struct Packed { uint256 b; uint64 a; uint64 c; } // 2 slots
    Unpacked public u;
    Packed public p;
    function setPacked(uint256 b_, uint64 a_, uint64 c_) external { p = Packed(b_, a_, c_); }
```

```
    function setUnpacked(uint64 a_, uint256 b_, uint64 c_) external { u = Unpacked(a_, b_, c_); }
}
```

3.2 Prefer calldata for Large, Read only Inputs

Avoid copying into memory when inputs are not mutated. The trade offs between `storage`, `memory`, and `calldata` are summarized in Table 3. When a function is `external` and its parameters are large arrays or strings, using `calldata` keeps reads cheap and avoids memory expansion.

Table 3. Data locations and typical usage

Attribute	storage	memory	calldata
Persistence	Permanent	Per call	Per external call
Mutability	Mutable	Mutable	Read only
Gas	Very high	Medium	Low
Typical use	Contract state	Local buffers	External inputs

Listing 5. Prefer `calldata` when inputs are read only

```
contract WhereDataLives {
    function sumCalldata(uint256[] calldata xs) external pure returns (uint256 s) {
        for (uint256 i = 0; i < xs.length; ) { s += xs[i]; unchecked { i++; } }
    }
    function sumMemory(uint256[] memory xs) external pure returns (uint256 s) {
        for (uint256 i = 0; i < xs.length; ) { s += xs[i]; unchecked { i++; } }
    }
}
```

3.3 Cache Storage Reads Inside Loops

Read a storage value once, then reuse the cached value. This is most visible with loop bounds and with fields that are referenced many times inside the same function.

Example (Listing 6). The cached version reads `arr.length` only once, which saves repeated reads on long arrays and keeps the loop body focused on per element work.

Listing 6. Cache storage to memory inside loops

```
contract CacheDemo {
    uint256[] public arr;

    function sumNaive() external view returns (uint256 s) {
        for (uint256 i = 0; i < arr.length; i++) {
            s += arr[i]; // SLOAD per element and repeated length checks
        }
    }

    function sumCached() external view returns (uint256 s) {
        uint256 len = arr.length; // cache length in memory
        for (uint256 i = 0; i < len; i++) {
            s += arr[i]; // still SLOAD per element, fewer length reads
        }
```

```
    }
}
```

3.4 Short Circuiting and Simple Loop Counters

Place the cheaper or more selective condition first. Use pre increment and simple comparisons when possible. These changes are small on their own, but they add up when they appear in hot loops.

Example (Listing 7). Only evaluate the second predicate when the first passes. This saves work when the first check fails often, which is common in access control and bounds checks.

Listing 7. Short circuit evaluation pattern

```
require(isAllowed(user) && hasSufficientBalance(user), "not allowed or no balance");
```

3.5 Safe Use of unchecked

Disable overflow checks only with a proof that overflow cannot occur. The advantage is a small saving per arithmetic operation, which becomes visible in tight loops. The risk is reintroducing integer bugs, so limit usage to counters with clear bounds.

Example (Listing 8). The loop counter cannot overflow because it is bounded by length. This pattern is simple to audit and keeps the benefit local to the loop.

Listing 8. Safe use of unchecked for loop counters

```
contract UncheckedDemo {
    uint256[] public data;
    function sum() external view returns (uint256 s) {
        uint256 n = data.length;
        for (uint256 i = 0; i < n; ) { s += data[i]; unchecked { i++; } }
    }
}
```

3.6 Mapping versus Array Trade offs

Mappings give direct access without iteration. Arrays are iterable but updates can be costly. If you need both iteration and fast lookup, maintain a mapping for data and a separate array of keys. This hybrid layout makes iteration explicit and keeps per key reads at $O(1)$ while acknowledging that maintaining the key list is itself a write.

4 MEASURING AND PROFILING GAS USAGE

Optimization is empirical. Measure first, then change code. The workflow is to write a small hypothesis, run the test suite with a gas report, and keep changes that are supported by data. This keeps style debates grounded in numbers and prevents accidental regressions.

4.1 Reading Execution Traces

Frameworks can emit traces with per call gas. In Foundry, increase verbosity and inspect the call tree. Focus on storage operations and memory expansion because they dominate cost. When a line is unexpectedly expensive, check whether it touches new storage slots or writes at high memory offsets, then adjust the design and remeasure.

4.2 Gas Reports and Snapshots

Automate measurement in tests. Foundry prints a gas report with `forge test --gas-report`. Hardhat can use `hardhat-gas-reporter`. Record a baseline and compare in CI. In Foundry, `forge snapshot` produces a `.gas-snapshot`. Use `forge snapshot --check` to fail on regressions so that increases are reviewed before merge.

5 UNIT TESTING SMART CONTRACTS

Unit tests verify specific behavior in isolation. Use the Arrange, Act, Assert pattern and keep each test small so failures point to one idea at a time. Isolate tests with fresh deployments or state snapshots to avoid hidden dependencies between cases.

5.1 Expected Reverts

Negative tests must assert that invalid calls fail. This documents the boundary of valid behavior and guards against accidental expansion of privileges. It also improves the signal in code review since the tests state which paths must not succeed.

Example (Listing 9). Expect a revert with a specific message. The call immediately communicates which precondition is being validated.

Listing 9. Asserting expected reverts in Foundry

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
import "forge-std/Test.sol";
import "src/MyContract.sol";

contract MyContractTest is Test {
    MyContract c;
    address owner = address(0x1);
    address notOwner = address(0x2);

    function setUp() public { vm.prank(owner); c = new MyContract(); }

    function test_RevertWhen_CallerIsNotOwner() public {
        vm.prank(notOwner);
        vm.expectRevert("Unauthorized");
        c.protectedFunction();
    }
}
```

5.2 Event Assertions

Events communicate outcomes to off chain code. Tests that check event contents ensure that indexers and front ends receive the right signals. This is especially important when state is derived off chain from logs.

Example (Listing 10). Check that `Transfer` is emitted with expected fields. The template emissions make intent explicit and keep the test readable.

Listing 10. Checking event emission

```
function test_TransferEvent() public {
    address to = address(0x3);
```

```
    uint256 amount = 100 ether;
    vm.expectEmit(true, true, false, true);
    emit Transfer(address(this), to, amount);
    myToken.transfer(to, amount);
}
```

6 PROPERTY BASED TESTING AND INVARIANTS

Properties express truths that must hold for any valid input. Foundry treats any test with parameters as a fuzz test and reports counterexamples. This complements unit tests, which check examples, by searching the input space automatically. When a counterexample appears, shrink it to a minimal case and add a targeted unit test that reproduces the bug.

6.1 Examples versus Properties

A property describes a behavior that should hold across inputs. Examples confirm a single case. Both are useful, and together they outline the surface and the interior of the behavior you intend.

Example (Listing 11). A unit example and a commutativity property. The property does not care which numbers are provided because the relation must hold for all.

Listing 11. Examples versus properties

```
function test_Add_Example() public { assertEq(calculator.add(2,3), 5); }

function testFuzz_Add_Commutative(int256 a, int256 b) public {
    assertEq(calculator.add(a,b), calculator.add(b,a));
}
```

6.2 Constraining Inputs and Shrinking

Discard inputs that are not relevant to the property to focus the search. Constraints reduce time spent on trivial cases and increase the chance of reaching interesting states. When a failure is found, the fuzzer shrinks inputs to a simpler counterexample that is easier to understand.

Example (Listing 12). Skip trivial zero amounts. This keeps attention on the branch where value actually moves.

Listing 12. Constraining fuzz inputs

```
function testFuzz_Withdraw(uint96 amount) public {
    vm.assume(amount > 0);
    // ... test body ...
}
```

6.3 Designing Useful Invariants

For ERC20, conserve total supply and maintain balance sheet equality. For escrow, preserved value equals the sum of credits, and withdrawals cannot exceed deposits. These invariants are short to state yet powerful to test, and they generalize across many implementations.

7 TOOLCHAIN OVERVIEW

A layered toolchain improves correctness and repeatability. A compact comparison appears in Table 4. In practice the framework runs compilation and tests, the analyzers scan source for smells,

and the debuggers simulate transactions to explain failures. Integrating these steps in CI turns them into a routine gate rather than an optional activity.

Table 4. Common tools in a smart contract workflow

Tool	Category	Primary Function
Foundry	Dev framework	Compile, test, fuzz, gas report, deploy with Solidity tests
Hardhat	Dev framework	Compile, test with JS/TS, plugins, scripts
Remix	Web IDE	Rapid prototyping and debugging in browser
Slither	Static analyzer	Detects vulnerabilities and gas smells from source
Echidna	Fuzzer	Property based testing with custom invariants
Mythril	Symbolic exec	Explores paths to find complex bugs
Tenderly	Debug platform	Simulation, traces, gas profiler, monitoring

8 SUMMARY AND KEY TAKEAWAYS

- Gas is the main resource that shapes design. Reduce storage writes, prefer warm accesses, and avoid unnecessary memory expansion. When in doubt, search for `SSTORE` and cold `SLOAD` in a trace first since they explain most of the cost.
- Tests should include expected reverts, event checks, and property based fuzzing. Keep tests isolated and small so that failures localize quickly and test names read as documentation for intended behavior.
- Measure before optimizing. Use traces, gas reports, and CI snapshots to prevent regressions. Keep a short changelog of cost relevant decisions so that future edits do not undo earlier gains.
- Use a layered toolchain. Combine a development framework with static analysis, fuzzing, and path exploration. Monitor live behavior with simulation and profiling so that incidents can be reproduced and explained without guesswork.

REFERENCES