Synthetic Smart Contract Workloads for Testnet Evaluation

This document describes three synthetic Ethereum smart contracts that are designed to generate distinct types of workload on a local or development network. All three contracts are intentionally simple from a business-logic perspective: their purpose is not to implement useful application logic, but to allow controlled stress tests and measurements.

The three contracts are:
- HighGasContract – storage-heavy, very expensive transactions.
- HighComputeContract – compute- and memory-heavy transactions with minimal state growth.
- LowGasSpamContract – very cheap transactions suitable for high-volume spam.

Although they share the common goal of generating synthetic load, each contract targets a different bottleneck in the execution and networking pipeline. This makes them suitable for a comparative study in a thesis or research paper.

1. Contract Overviews and Goals

1.1 HighGasContract

HighGasContract is designed to maximise the gas cost per transaction by combining several of the most expensive EVM operations in a tight loop: storage reads and writes (SLOAD and SSTORE), cryptographic hashing (KECCAK256) and event emission (LOGs). The main entrypoints, burnGas(uint256 iterations) and burnStorageOnly(uint256 iterations), allow the caller to scale the amount of work per transaction.

In burnGas, each loop iteration performs the following pattern:
- Reads a value from a mapping<uint256, uint256> (SLOAD).
- Computes a new keccak256 hash over the current value, a global accumulator and block metadata.
- Writes the resulting value back into the mapping (SSTORE), which is particularly expensive when writing to new slots or changing zero to non-zero.
- Updates the global accumulator sink, ensuring that the computation has observable side effects.
- Emits an event DidWork(...) with the loop index, hash and written value.

The burnStorageOnly variant focuses even more on storage churn by writing to many different mapping keys, which accelerates state trie growth. Overall, HighGasContract represents a storage-heavy, high-gas workload that stresses block execution time and state growth.

1.2 HighComputeContract

HighComputeContract focuses on CPU and memory usage rather than storage. It provides two functions, burnCpuAndMemory(uint256 iterations, uint256 arraySize) and burnHashOnly(uint256 iterations). The first allocates a large in-memory array and fills it inside nested loops using repeated keccak256 hashing. This triggers substantial memory expansion and CPU load within a single transaction.

By design, the contract touches storage only once at the end of the function, where it stores an aggregated result in the sink variable and emits a ComputeWork event. This means that the chain state barely grows, even if many transactions are executed. HighComputeContract therefore serves as a compute- and memory-heavy workload with minimal state bloat.

1.3 LowGasSpamContract

LowGasSpamContract is intentionally designed to minimise the gas cost per transaction. Its primary function spam(uint64 tag) emits a small event and performs no storage reads, no storage writes and no hashing. As a result, most of the gas cost per transaction comes from the intrinsic transaction cost and a single LOG operation.

A second function, multiSpam(uint64[] calldata tags), emits multiple events in a loop but still avoids any storage access. In practice, the simple spam function is ideal for high-volume transaction spam, as it allows experiments to push the network to very high transaction throughput without rapidly increasing the on-chain state size.

2. Comparative Analysis

Although the three contracts share the overarching purpose of generating synthetic workloads on an Ethereum-compatible network, their operational profiles are deliberately different.

Overlap (cross-section):

- All three are simple state-changing contracts that can be invoked with straightforward transactions from tools such as Spamoor.
- All three emit events, ensuring that each transaction produces a log entry that can be observed in receipt data.
- None of them implements complex business logic; they are micro-benchmarks focused on resource consumption characteristics rather than application semantics.

Key differences:

- Resource focus:
  - HighGasContract: storage-heavy (SSTORE/SLOAD), plus hashing and logging.
  - HighComputeContract: compute- and memory-heavy (keccak256 and memory expansion).
  - LowGasSpamContract: minimal execution, minimal storage, small logs only.
- State impact:
  - HighGasContract: rapid and significant state growth.
  - HighComputeContract: almost no state growth (one storage slot updated).
  - LowGasSpamContract: no state growth at all.
- Typical gas per transaction:
  - HighGasContract: very high, often near block gas limits for moderate iteration counts.
  - HighComputeContract: high, but dominated by computation and memory.
  - LowGasSpamContract: low, dominated by intrinsic transaction cost and a single LOG.
- Intended experimental use:
  - HighGasContract: evaluate client behaviour under storage-intensive load, state growth and heavy block execution.
  - HighComputeContract: evaluate raw EVM performance without significant state growth.
  - LowGasSpamContract: evaluate throughput, mempool, networking and consensus under high transaction volume.

2.1 Comparison Table

Table 1 summarises the main characteristics and experimental goals of the three contracts. It can be directly reused in the thesis (converted to LaTeX or another format if required).

Contract	Main resource focus	State impact	Typical gas / tx	Primary experimental
HighGasContract	Storage + hashing + logs	High, rapid state growth	Very high	Storage-heavy blocks, sta
HighComputeContract	CPU (hash) + memory	Very low (1 slot)	High	Raw EVM compute/memo
LowGasSpamContract	Tx count + logs only	None	Low	Throughput, mempool and

3. Detailed Code Explanations

3.1 HighGasContract

HighGasContract combines expensive storage operations (SSTORE and SLOAD), hashing and event emission in tight loops to create a storage-heavy, high-gas workload.

Key storage layout:

- uint256 public sink: a global accumulator that changes on every call. It prevents the compiler from treating the computation as dead code and provides an observable state change between transactions.
- mapping(uint256 => uint256) public store: a mapping used to generate many distinct storage slots. Repeated writes into this mapping cause the underlying state trie to grow.

The main function burnGas(uint256 iterations) executes the following pattern in each loop iteration:

1) SLOAD: it reads store[i], incurring the gas cost of a storage read.
2) Hashing: it computes a keccak256 hash over the current value, sink and block metadata. This stresses the EVM's built-in hash function.
3) SSTORE: it writes a derived value back to store[i]. Writing to storage is one of the most expensive operations in the EVM, especially when writing to new slots or changing zero to non-zero.
4) Accumulator update: it updates sink using XOR with the new value, ensuring that the global state changes in a data-dependent way.
5) Event: it emits DidWork(i, hashResult, valueWritten), adding a LOG operation and log data to the receipt.

The burnStorageOnly(uint256 iterations) variant focuses on state growth by writing to many different keys (such as i + sink). This pattern quickly increases the number of storage slots in use and stresses disk I/O and trie updates even more. Together, these functions make HighGasContract an effective tool for studying storage-heavy workloads.

3.2 HighComputeContract

HighComputeContract is designed so that most of its gas cost comes from computation and memory usage, not from storage. It uses a similar sink accumulator to HighGasContract but touches storage only once per call.

In burnCpuAndMemory(uint256 iterations, uint256 arraySize), the function first checks that the parameters are greater than zero. It then initialises a local hash variable h and an accumulator acc, and allocates a bytes32[] memory array of size arraySize. Memory allocation triggers EVM memory expansion, which becomes increasingly expensive as the memory size grows.

The core work is done in a nested loop:

- Outer loop (iterations): controls how many times the inner loop and subsequent hash are executed.

• Inner loop (arraySize): fills each array slot with a new keccak256 hash over the previous hash, the accumulator, the loop indices and block metadata. Each inner iteration writes to memory and updates the accumulator using XOR.
After finishing an inner loop, the function hashes over h, data.length and acc, forcing additional hashing and memory reads. At the end, it writes a combination of acc and h to sink (one SSTORE) and emits a ComputeWork event.

The burnHashOnly(uint256 iterations) function provides a simpler workload that only performs a loop of keccak256 operations without the large memory array. It is useful as a lower-gas baseline or for isolating the cost of hashing alone.

3.3 LowGasSpamContract

LowGasSpamContract is a deliberately minimal contract intended for high-volume transaction spam at the lowest possible gas cost per transaction.

The contract defines a single event Spam(address indexed sender, uint64 tag). The primary function, spam(uint64 tag), simply emits this event using msg.sender and the supplied tag. There are no storage reads or writes, no loops and no hashing. Consequently, the gas cost of the function is dominated by the intrinsic transaction cost and the single LOG operation needed to emit the event.

The multiSpam(uint64[] calldata tags) function emits multiple Spam events in a loop, still without touching storage. It is slightly more expensive per transaction than spam but can be used to compare workloads with one event per transaction versus multiple events per transaction.

Because neither function modifies storage, the chain state does not grow regardless of how many spam transactions are executed. This makes LowGasSpamContract ideal for experiments that focus on mempool behaviour, transaction throughput, block propagation and consensus, independently of state size.

4. Solidity Source Code

4.1 HighGasContract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

/// @notice Extremely gas-expensive contract for benchmarking / spam testing.
///         DO NOT use on mainnet. Intended only for local / test networks.
contract HighGasContract {
    // State used to force SSTORE and prevent optimizer from pruning work.
    uint256 public sink;
    mapping(uint256 => uint256) public store;

    event DidWork(uint256 indexed i, bytes32 hashResult, uint256 valueWritten);

    /// @notice Burns a lot of gas by doing:
    ///         - SLOAD on a mapping
    ///         - KECCAK256 hashing
    ///         - SSTORE (expensive, especially zero -> non-zero)
    ///         - LOG/emit event
    /// @param iterations Number of loop iterations. Keep it small or tx will OOG.
    function burnGas(uint256 iterations) external {
        // Tight loop with multiple expensive operations.
        for (uint256 i = 0; i < iterations; ++i) {
            // 1) Expensive SLOAD
            uint256 current = store[i];
```

```
            // 2) Expensive KECCAK256 hash
            bytes32 h = keccak256(
                abi.encodePacked(
                    current,
                    sink,
                    block.timestamp,
                    block.number,
                    i
                )
            );

            // 3) Expensive SSTORE (zero -> non-zero especially costly)
            uint256 newValue = uint256(h) ^ (current + 1);
            store[i] = newValue;

            // 4) Update global sink so optimizer can't drop the work
            sink ^= newValue;

            // 5) Expensive LOG (event) with topics and data
            emit DidWork(i, h, newValue);
        }
    }

    /// @notice Variant that focuses more on storage churn (lots of SSTORE).
    ///         This can be even nastier for state growth.
    function burnStorageOnly(uint256 iterations) external {
        for (uint256 i = 0; i < iterations; ++i) {
            // Each write to a *new* slot or zero->non-zero costs a lot of gas.
            uint256 newValue = uint256(
                keccak256(abi.encodePacked(i, sink, blockhash(block.number - 1)))
            );

            store[i + sink] = newValue;
            sink += newValue; // keep changing sink so the index and values evolve
        }
    }
}
```

4.2 HighComputeContract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

/// @notice Gas-expensive contract that focuses on CPU + memory usage
///         instead of heavy storage writes. For local / test networks only.
contract HighComputeContract {
    // Used to prevent optimizer from pruning work and to carry results across calls.
    uint256 public sink;

    event ComputeWork(bytes32 finalHash, uint256 iterations, uint256 arraySize, uint256 sinkValue);

    /// @notice Burns gas by:
    ///         - Allocating a large memory array (memory expansion)
    ///         - Running tight nested loops
    ///         - Repeated KECCAK256 hashing over changing data
    ///
    /// @param iterations  Outer loop count (CPU emphasis).
    /// @param arraySize   Size of the in-memory array (memory emphasis).
    function burnCpuAndMemory(uint256 iterations, uint256 arraySize) external {
        require(iterations > 0, "iterations must be > 0");
        require(arraySize > 0, "arraySize must be > 0");

        bytes32 h = bytes32(0);
        uint256 acc = sink;
```

```
        // Allocate a big chunk of memory once (triggers memory expansion).
        bytes32[] memory data = new bytes32[](arraySize);

        for (uint256 i = 0; i < iterations; ++i) {
            // Fill the array with hashes depending on i, j, and acc.
            for (uint256 j = 0; j < arraySize; ++j) {
                // Heavy hashing work
                h = keccak256(abi.encodePacked(h, acc, i, j, block.timestamp, block.number));
                data[j] = h;

                // Mix into accumulator so the optimizer can't remove this.
                acc ^= uint256(h);
            }

            // Hash the whole array (forces reading from memory).
            h = keccak256(abi.encodePacked(h, data.length, acc));
        }

        // Persist something to storage so that work is observable from outside.
        sink = acc ^ uint256(h);

        emit ComputeWork(h, iterations, arraySize, sink);
    }

    /// @notice A lighter helper that only does hashing without the inner array loop.
    ///         Useful as a lower-gas baseline vs burnCpuAndMemory.
    function burnHashOnly(uint256 iterations) external {
        require(iterations > 0, "iterations must be > 0");

        bytes32 h = bytes32(sink);
        uint256 acc = sink;

        for (uint256 i = 0; i < iterations; ++i) {
            h = keccak256(abi.encodePacked(h, acc, i, block.timestamp, block.prevrandao));
            acc ^= uint256(h);
        }

        sink = acc ^ uint256(h);
        emit ComputeWork(h, iterations, 0, sink);
    }
}
```

4.3 LowGasSpamContract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

/// @notice Low-gas transaction spam contract.
///         Each call is a valid tx but does almost nothing:
///         just emits a tiny event. Perfect for high-TPS spam
///         with minimal gas per tx on a testnet.
contract LowGasSpamContract {
    /// @dev Minimal event: 1 indexed topic (sender) + 1 small value.
    event Spam(address indexed sender, uint64 tag);

    /// @notice Cheapest spam entry point.
    /// @param tag Arbitrary small number (e.g. random, counter from the spammer).
    ///            Lets you distinguish txs in logs if needed.
    function spam(uint64 tag) external {
        // No storage writes, no loops, no hashing – just an event.
        emit Spam(msg.sender, tag);
    }

    /// @notice Slightly more expensive variant:
```

```
    ///         emits multiple events in a single tx (still no storage).
    ///         Use only if you want to compare "many tx" vs "more work per tx".
    function multiSpam(uint64[] calldata tags) external {
        for (uint256 i = 0; i < tags.length; ++i) {
            emit Spam(msg.sender, tags[i]);
        }
    }
}
```

5. Practical Considerations for Experiments

When using these contracts in experiments, several practical aspects should be considered:

- Parameter selection: both HighGasContract and HighComputeContract expose an iterations parameter (and arraySize for HighComputeContract). These parameters must be tuned so that transactions consume a large, but not catastrophic, fraction of the block gas limit. This allows meaningful measurements without frequent out-of-gas failures.
- Measuring different layers: by combining these contracts, experiments can decouple execution-layer bottlenecks from networking-layer bottlenecks. For example, comparing a HighGasContract workload to a LowGasSpamContract workload at the same block fill level can show whether node performance is dominated by computation or by transaction volume.
- State management: repeated HighGasContract calls will rapidly increase the on-disk state size. For longer-running experiments, it may be useful to reset the chain periodically or to measure how pruning, snapshots or state-sync strategies behave.
- Reproducibility: because the contracts are deterministic given the same inputs, they are well-suited for controlled, repeatable benchmarks. Recording the exact parameters (iterations, arraySize, transaction rate) is important for reproducing results.

6. Conclusion

The three synthetic contracts—HighGasContract, HighComputeContract and LowGasSpamContract—form a compact but expressive suite of workloads for evaluating an Ethereum-compatible network. They share a common interface style and purpose, yet their gas and state profiles are intentionally distinct.

Together, they allow a researcher to:

- Stress-test storage-heavy scenarios with rapid state growth (HighGasContract).
- Isolate compute and memory performance of the EVM (HighComputeContract).
- Investigate high-throughput, low-cost transaction spam and networking behaviour (LowGasSpamContract).

This separation of concerns is valuable in a thesis context, because it enables a clear explanation of which part of the system is being tested by each experiment and why the observed performance differences occur.