# Custody Protocol — ABCI++ Callback Mapping, RocksDB Batching, and Executor Interface

You are targeting ABCI++ (CometBFT). In this model, CometBFT proposes blocks and runs consensus; your application is a deterministic state machine.
The key architectural change vs classic ABCI is: block execution happens in FinalizeBlock, and persistence happens in Commit.

This document maps:
1) exact ABCI++ callbacks and responsibilities
2) the cleanest RocksDB batching approach (block-scoped overlay + atomic commit)
3) how to shape your execution engine interface for ABCI++

## 1) ABCI++ callback flow (what gets called, and when)

| Callback | Purpose | Your PoC Responsibility |
|---|---|---|
| CheckTx | Mempool admission (per tx) | Decode + signature verify + strict nonce sanity (optional) + basic structu |
| PrepareProposal | Proposer chooses txs to include | Select txs (usually pass-through) enforcing max bytes / tx count. Optiona |
| ProcessProposal | Validators accept/reject proposed block | Fast deterministic validation: decode+sig, size limits, reject malformed tx |
| FinalizeBlock | Execute the block deterministically | Create a block-scoped state overlay; execute txs sequentially; produce t |
| Commit | Persist state for the accepted block | Atomically write staged batch to RocksDB; update app_hash; clear block |
| Query | Read-only state queries | Serve from last committed state (PoC). Do not read uncommitted block d |
| ExtendVote | Vote extensions (optional feature) | Return empty for PoC unless you need them. |
| VerifyVoteExtension | Verify vote extensions | Return OK for empty extensions in PoC. |

Key point: FinalizeBlock may be run on proposals that do not ultimately commit (e.g., rejected proposals). Therefore, treat FinalizeBlock as producing a staged state transition (overlay), and Commit as the only place that persists changes to RocksDB.

## 2) Clean RocksDB batching approach for ABCI++

### 2.1 Requirements

Your state layer must support:
• read-your-writes within a block (tx #5 sees tx #2 writes)
• atomic commit at end of block
• deterministic iteration order for scans (lexicographic keys)
• zero reliance on wall-clock time (use ctx time/height)

Because you use strict nonces, nonce enforcement is consensus-critical and must be applied within block execution (FinalizeBlock), not just CheckTx.

## 2.2 Recommended approach: Snapshot + WriteBatchWithIndex

Use a RocksDB Snapshot plus WriteBatchWithIndex to implement a block overlay without hand-rolling a shadow map.

Pattern:
• At FinalizeBlock start: take a DB snapshot (consistent view of committed state)
• Create WriteBatchWithIndex (overlay)
• Implement KV::get as “overlay first, else snapshot DB”
• Implement KV::put/del into overlay
• At Commit: write the underlying WriteBatch to DB atomically

```
struct BlockKV {
  rocksdb::DB* db;
  const rocksdb::Snapshot* snap;
  rocksdb::ReadOptions ro;
  rocksdb::WriteBatchWithIndex wb;   // overlay (read-your-writes)

  BlockKV(rocksdb::DB* db_)
   : db(db_), snap(db->GetSnapshot()), wb(/*overwrite_key=*/true) {
      ro.snapshot = snap;
   }

  ~BlockKV() { db->ReleaseSnapshot(snap); }

  std::optional<Bytes> get(const Bytes& key) {
    std::string val;
    rocksdb::Status s =
      wb.GetFromBatchAndDB(db, ro, toSlice(key), &val);
    if (s.IsNotFound()) return std::nullopt;
    if (!s.ok()) throw ...;
    return Bytes(val.begin(), val.end());
  }

  void put(const Bytes& key, const Bytes& value) {
    wb.Put(toSlice(key), toSlice(value));
  }

  void del(const Bytes& key) {
    wb.Delete(toSlice(key));
  }
};
```

## 2.3 Prefix scans (approvals/attestations)

You have prefix scans for IA|intent| and AT|ws|subject|claim|. Implementing overlay-aware scans is doable but extra work.

Pragmatic PoC rule (recommended):
• approvals: rely on IntentState.approvals_count updated at approval-time (no need to scan during execution)
• attestations: assume attestations exist prior to the block that executes an intent (operationally common), or add overlay-aware scan later

If you want overlay-aware scan later, implement a deterministic merge:
• iterate committed DB prefix in lexicographic order

• iterate overlay entries for the same prefix
• merge into one sorted stream applying deletes/overwrites from overlay

## 2.4 Commit behavior

In Commit:
• Persist the staged overlay to RocksDB using a single DB::Write call
• Update app_hash (32 bytes). With no Merkle tree, use a deterministic hash chain (BLAKE3) over prior app_hash + block digest + tx-results digest.
• Clear the overlay and any cached block context

```
// App member state (lives across callbacks)
std::unique_ptr<BlockKV> g_block;     // active only between FinalizeBlock and Commit
Hash32 g_app_hash;                   // last committed app hash

CommitResponse Commit() {
  rocksdb::WriteOptions wo;
  wo.sync = true; // PoC: durable
  db->Write(wo, g_block->wb.GetWriteBatch());

  g_app_hash = blake3("APPHASH" || g_app_hash || last_block_id || tx_results_digest);

  g_block.reset();
  return { .data = g_app_hash_bytes };
}
```

# 3) How this changes your execution engine interface

## 3.1 New executor interface (recommended)

Make your executor explicitly block-scoped by passing a KV overlay into apply_tx.
Your existing handler structure remains the same; only the storage handle changes from “DB” to “BlockKV overlay”.

```
struct TxResult {
  uint32_t code;           // 0 = OK
  std::string log;
  std::vector<Event> events;
};

struct TxContext {
  uint64_t now_ms;         // from block time
  uint64_t height;
  Hash32 chain_id;
};

class Executor {
public:
  // consensus-critical state transition (no IO except through KV)
  TxResult apply_tx(const TxEnvelopeV1& tx, BlockKV& kv, const TxContext& ctx);
};

FinalizeBlockResponse FinalizeBlock(const Request& req) {
  g_block = std::make_unique<BlockKV>(db);  // snapshot + overlay
  std::vector<TxResult> results;

  for (auto& raw_tx : req.txs) {
    auto tx = decode_and_verify(raw_tx);    // sig + envelope checks
    auto r  = executor.apply_tx(tx, *g_block, ctx_from(req));
    results.push_back(r);
  }

  // Do NOT write to DB here. Return tx results & events.
  // Keep g_block alive until Commit.
  return make_finalize_response(results);
}
```

## 3.2 Where strict nonces live

Because you require strict nonce ordering (+1), nonce enforcement MUST be part of apply_tx (FinalizeBlock execution).
CheckTx may reject obvious mismatches for mempool hygiene, but FinalizeBlock is the consensus source of truth.

## 3.3 PrepareProposal / ProcessProposal guidance (PoC)

For a custody PoC:
• PrepareProposal: pass through txs up to max bytes/tx count; optionally drop malformed txs
• ProcessProposal: accept unless malformed bytes or invalid signatures
Do not attempt state transitions here; all state transitions happen in FinalizeBlock.

## 4) Fit with your chosen invariants

These ABCI++ patterns fit your locked invariants cleanly:
• BLAKE3 everywhere
• strict nonce +1 enforced in apply_tx
• snapshotted policy for approval/execution
• intent index (IV|ws|vault|created_at_ms|intent_id) written atomically in ProposeIntent

All are executed on the block overlay in FinalizeBlock and persisted once in Commit.