CORE BANKING MVP

Focus Group Report

Challenging Every Assumption Before Writing a Single Line of Code

Giovanni Everduin
Dubai to Paris - DXB-CDG - May 2026
Conducted at the gate during a 1-hour takeoff delay

Assumptions Challenged	**7**
Decisions Informed	**41**
Layers Built	**7 / 7**
Tests Passing	**106**
Lines of TypeScript	**~6,000**

Executive Summary

Before writing a single line of code for the core banking MVP, we ran a structured exercise to surface and challenge the inherited assumptions that most banking systems carry as unexamined baggage. The goal was not to reject tradition for the sake of novelty - it was to identify which conventions are fundamentally necessary and which are simply the residue of decisions made decades ago under different constraints.

The exercise took place at Dubai International Airport during a one-hour takeoff delay. Seven core assumptions about how banking systems work were identified, examined, and either validated or replaced with a first-principles alternative. Every assumption that was replaced directly informed at least one architectural decision in the system that was subsequently built during the flight.

The result was a set of design principles that are not opinions - they are conclusions derived from asking what is provably true about how money moves, how value is recorded, and how systems prove their own integrity. These principles became the foundation for 41 architectural decisions and 7 layers of working code.

Why a Focus Group of Assumptions?

Most core banking systems are built by adding features to an existing architecture. The architecture itself is rarely questioned because it works - or at least, it appears to work. But the gap between 'appears to work' and 'provably correct' is where most banking incidents live: overnight reconciliation that discovers a discrepancy twelve hours too late, floating-point arithmetic that silently rounds away fractions of a cent, settlement systems bolted on as afterthoughts that require their own reconciliation layer to trust.

The premise of this exercise was simple: if you were building a bank's core from scratch today, with no legacy constraints, no inherited vendor decisions, and no regulatory obligation to use any specific architecture - what would you actually build? What assumptions would survive contact with first principles?

> *"The question the industry keeps avoiding: how hard is this really?"*

Methodology

Each assumption was evaluated through a three-step process:

1. Identify the convention - what does the industry take for granted?

2. Ask the first-principles question - is this fundamentally true, or just inherited?

3. If inherited, design the replacement - what would you build if you started from zero?

The replacement was not accepted unless it could be expressed as a testable invariant. 'Events are truth' is not just a philosophy - it is a property that can be verified by replaying the event log and comparing the result against any stored state. If they diverge, the stored state is wrong. Always.

The Seven Assumptions

ASSUMPTION 01

~~State is stored in database rows~~

Events are truth. State is derived.

The Convention

Traditional core banking systems store account balances as mutable rows in a database. When money moves, the system updates two rows - debit one, credit another. The balance you see is the balance that exists. There is no way to independently verify it without a separate audit trail.

The Problem

This creates a system where the current state is authoritative but unverifiable. If a row is corrupted, there is no mechanism to detect it unless a separate reconciliation process catches the discrepancy - usually hours or days later. The state and the history are decoupled, which means they can diverge.

The Replacement

In an event-sourced system, the event log is the source of truth. Balances are never stored - they are computed by replaying the events. If you need the balance of account X, you read every event that affected account X and calculate. The computation is deterministic. The result is provably correct. If a projection cache exists, it is disposable - the log is not.

`Informed: Decisions 09 (event versioning), 12 (JSON payload storage), 13 (projection purity), 14 (optimistic concurrency)`

ASSUMPTION 02

~~Reconciliation runs overnight in batch~~

Continuous. Real-time. Always.

The Convention

Batch reconciliation is a product of an era when compute was expensive and databases locked under load. Running a full reconciliation across all accounts was something you did once a day, after hours, when the system was quiet. Discrepancies discovered at 2 AM were investigated the next morning.

The Problem

A 12-hour delay between a discrepancy occurring and being detected is not a feature - it is a risk. In the worst case, a systematic error compounds across thousands of transactions before anyone notices. The batch model also creates a false sense of security: if reconciliation passed last night, the assumption is that everything is fine today. It might not be.

The Replacement

Reconciliation is continuous. Any account balance can be verified against the full event history at any moment. A SHA-256 hash chain on the event log provides tamper detection - each event's hash includes the previous event's hash. If any event in the chain is modified, the chain breaks and the system flags the exact divergence point.

`Informed: Decision 18 (per-currency trial balance), Layer 4 (reconciliation engine)`

ASSUMPTION 03

~~Settlement is a separate system~~

Settlement is native to the core.

The Convention

In most banks, settlement is handled by a completely separate system - often from a different vendor, running on different infrastructure, with its own data model. The core banking system hands off transactions to the settlement system and hopes for the best. A separate reconciliation layer sits between them to catch mismatches.

The Problem

This architecture exists because settlement was historically complex, slow, and operated on different rails (SWIFT, SEPA, local clearing houses). But the complexity was in the rails, not in the concept. Settlement is just confirmation that a value transfer has been finalized. Treating it as a separate system adds integration complexity, reconciliation overhead, and failure modes.

The Replacement

Settlement is a pluggable adapter inside the core. The SettlementAdapter interface defines initiate(), confirm(), getStatus(), and cancel(). A mock adapter simulates T+0,

T+1, and T+2 settlement cycles. An EVM adapter, a Solana adapter, or a SWIFT adapter can be swapped in without changing a single line of core logic. The interface is the product.

`Informed: Decision 04 (mock adapter in MVP, EVM interface designed)`

ASSUMPTION 04

~~The audit trail is a log file somewhere~~

The event log IS the audit trail.

The Convention

Most systems treat auditing as a cross-cutting concern - something bolted on via logging middleware, written to a separate database, and queried through a separate reporting tool. The audit trail and the operational data are maintained independently.

The Problem

Independent maintenance means independent failure modes. The audit trail can miss events if the logging middleware fails. The audit trail can disagree with the operational data if there is a race condition. And verifying the audit trail against the operational data requires yet another reconciliation layer.

The Replacement

In an event-sourced system, there is no separate audit trail because the event log serves both purposes. Every state change is an event. Every event is immutable and append-only. The operational state and the audit history are the same data structure. They cannot diverge because they are not separate.

`Informed: Decisions 12 (event payload format), 14 (optimistic concurrency prevents overwrites)`

ASSUMPTION 05

~~Currency means fiat. Period.~~

Fiat, stablecoin, CBDC, token. All native.

The Convention

Traditional core banking systems were built for fiat currencies - USD, EUR, GBP, AED. The data model assumes currency is a three-letter ISO code and that all currencies behave the same way. Digital assets, stablecoins, and CBDCs were not part of the original design.

The Problem

Retrofitting multi-asset support into a fiat-only system is expensive and error-prone. CBDCs have currency properties but also asset-class characteristics - issuance rules, programmability constraints, jurisdiction-specific behavior. Treating a digital dirham exactly like a physical dirham misses constraints that matter.

The Replacement

The system is currency-aware from day one. The Money value object carries both amount and currency. Mixed-currency operations throw CurrencyMismatchError - they never silently coerce. CBDCs are modeled as an asset class with currency-like behavior, future-proofing for multi-CBDC environments. Trial balances run per-currency because books must balance independently in each denomination.

`Informed: Decisions 02 (multi-currency from day one), 05 (CBDC as asset class), 11 (throw on mixed currency), 18 (per-currency trial balance)`

ASSUMPTION 06

~~Business logic lives in middleware~~

Logic lives at the ledger layer.

The Convention

The middleware layer in traditional banking architecture handles everything from overdraft checks to compliance screening to fee calculation. It sits between the API and the core, intercepting transactions and applying rules. The core itself is treated as a dumb ledger.

The Problem

Middleware becomes a single point of complexity. Rules are scattered across interceptors, plugins, and configuration files. Testing requires standing up the full middleware stack. And the 'dumb ledger' core cannot enforce its own invariants - it trusts that middleware will never send it an invalid instruction.

The Replacement

Overdraft policy is a pluggable interface at the engine level - the engine enforces it, not middleware. Posting rules are pure functions that map business events to balanced journal entries. Compliance is event-driven: consumers subscribe to the event stream and act independently. The core never knows compliance exists. This means compliance rules can change without touching a single line of core code.

`Informed: Decisions 03 (pluggable overdraft), 06 (event-driven compliance), 15 (posting rules as projection), 16 (system accounts)`

ASSUMPTION 07

~~Floats are fine for money if you round correctly~~

Floats are never fine for money. Ever.

The Convention

JavaScript's Number type is an IEEE 754 double-precision float. It cannot exactly represent 0.1. The expression 0.1 + 0.2 evaluates to 0.30000000000000004. Most developers know this. Most banking systems still use floats somewhere in the stack, relying on rounding at the display layer to hide the imprecision.

The Problem

Rounding hides the problem; it does not fix it. Over millions of transactions, floating-point errors accumulate. A system that is 'close enough' on every individual transaction can be meaningfully wrong in aggregate. And the error is not deterministic - it depends on the order of operations, which means replaying transactions in a different order produces a different result.

The Replacement

All monetary arithmetic uses Decimal.js with 38-digit precision and banker's rounding (ROUND_HALF_EVEN). The Money value object wraps Decimal and exposes arithmetic methods that are type-safe and currency-safe. Comparisons use .eq(), .gt(), .lt() - never === or >. Amounts are stored in the event log as Decimal.toFixed() strings, ensuring perfect round-trip fidelity through JSON serialization.

`Informed: Decision 11 (throw on mixed currency, 38-digit precision, banker's rounding)`

Summary: Assumptions to Decisions

Old Assumption	New Reality	Decisions Informed
State is stored in database rows	Events are truth. State is derived.	09 (event versioning), 12 (JSON payload storage), 13
Reconciliation runs overnight in batch	Continuous. Real-time. Always.	18 (per-currency trial balance), Layer 4 (reconcilia
Settlement is a separate system	Settlement is native to the core.	04 (mock adapter in MVP, EVM interface designed)
The audit trail is a log file somewhere	The event log IS the audit trail.	12 (event payload format), 14 (optimistic concurrency
Currency means fiat. Period.	Fiat, stablecoin, CBDC, token. All native.	02 (multi-currency from day one), 05 (CBDC as asset
Business logic lives in middleware	Logic lives at the ledger layer.	03 (pluggable overdraft), 06 (event-driven compliance
Floats are fine for money if you round correctly	Floats are never fine for money. Ever.	01 (throw on mixed currency, 38-digit precision, ban

Conclusion

Seven assumptions challenged. Seven replacements designed. Each replacement expressed as a testable invariant in the codebase. The system does not just claim to be event-sourced or continuously reconciled - it proves these properties through 106 automated tests across 13 test files.

The most important outcome of this exercise was not the list of replacements - it was the discipline of refusing to build on unexamined foundations. Every line of code in the system can trace its design rationale back to a first-principles question asked at the gate before boarding.

> *"Whatever state it is in when we land is the honest answer to a question the industry keeps avoiding."*