Fintech Engineering Handbook

Patterns for building software that handles money

Voytek Pitula

Contents

Welcome to the Fintech Engineering Handbook. This resource aims to describe the most important patterns used in software engineering, where money is the primary focus of the system. It can be read in full to get a comprehensive understanding or in parts when dealing with a particular problem.

For whom?

- **People joining fintech.** To get familiar with the domain and the patterns that make money systems trustworthy.
- **People already in fintech.** As a reference to reach for when facing a particular problem, and a shared vocabulary to point colleagues at.
- **People outside fintech.** To understand how building for money differs from what they're used to, and why.

To learn more about the background of this book, see Appendix C.

Principles

Everything you will read below is a way to adhere to the three principles:

1. **No invented data.** Money can't be created out of nowhere, so we can't tolerate duplicates or arbitrary balance updates. We enforce this with idempotency, deduplication, and reconciliation.
2. **No lost data.** Everything that happens to money has to be tracked and persisted. We protect this with full precision, at-least-once deliveries, event sourcing, audit trails, and immutability.
3. **No trust.** Trust neither external providers, internal components, nor the world. We uphold this by verifying webhooks, cross-checking data across sources, and failing loudly on broken assumptions.

Representing money

Before you can move or record money, you have to represent it. These are the decisions about how a monetary value is modeled, stored, computed, and converted. Getting them wrong means every layer above inherits the error.

Precision handling

Money representation is one of the most fundamental decisions in financial systems. There are four primary ways to do it:

1. **Floating-point.** Built-in float or double types. This can create unpredictable precision losses and is almost never a good idea. But it's the fastest and most memory efficient, and requires no additional libraries or data structures.
2. **Arbitrary precision.** Types like Java's `BigDecimal` let you control the precision of a computation precisely. The code is predictable and we get to decide where and how rounding happens. It fits intermediate work like FX or pricing math, where many operations chain together.
3. **Minor-units precision.** For most fiat currencies it's ok to keep only a fixed precision, the same that is used in the connected central banking system. The number of digits is described by ISO 4217 (don't assume it's always 2, it's not!). In practice this means storing the amount as an integer in its smallest unit - €12.34 becomes `1234`. Crypto uses the same integer-smallest-unit idea (satoshis for BTC, wei for ETH), but with two twists: the precision is per-asset and defined by the token itself (e.g. an ERC-20′s `decimals`), often 18 digits, and the resulting magnitudes overflow 64-bit integers, so you need arbitrary-width integers to hold them.
4. **Rational numbers.** When no precision loss is acceptable. This is the most powerful approach but comes with its own caveats. First, it's slower than the alternatives. Second, it cannot be converted to other formats without losing precision. Third, it usually requires a custom datatype or a library.

Selecting one or the other depends on the class of the system and its responsibilities. There is no rule of thumb here, other than not using floating points. These representations are not mutually exclusive either - how you store an amount and how you compute with it are separate decisions, and a system often combines them, e.g. integer storage with `BigDecimal` for intermediate computation.

The same care applies when an amount is being serialized. A bare JSON number is an IEEE-754 double in most parsers, so serializing money as a number reintroduces the floating-point problem at the edge, no matter how carefully you represent it internally. Send money either as a string (`"12.34"`) or as an integer in its smallest unit.

Principles touched:

- **No lost data.** The wrong representation silently drops precision that can never be recovered.

Rounding strategies

1. **Rounding is inevitable.** It should be done explicitly: any division, currency conversion, fee, interest, or rate application, or move between precisions might require rounding.
2. **It's a business decision.** Different rounding strategies have different implications. Sometimes you have to be conservative (e.g. not to spend what you don't have) and round down; sometimes you care about statistical effects and use half-even. Deciding who gets the fraction might have legal/tax implications.

3. **Round as seldom as possible.** The longer you keep full precision, the more options you have to make the right decision in the right context. Rounding should usually happen on boundaries, e.g. before numbers are persisted or shown to the user.
4. **Rounding breaks sums.** If a number is split into parts and rounding is applied, the sum of the parts might no longer equal the original number. Depending on the context, this might require explicit handling - e.g. an explicit rounding account.

Principles touched:

- **No lost data.** Residuals must be tracked, not dropped.
- **No invented data.** Rounding must never mint money that wasn't there.

Currency handling

Money can't be represented as a number alone - it comes paired with a currency. There are a few nuances when it comes to handling currencies.

1. **Pack amount and currency together.** A `Money` newtype (struct, class, record, etc.) minimizes the chance of errors.
2. **No cross-currency arithmetic.** Your system should prohibit adding two amounts in different currencies. Conversion should happen very explicitly with a strictly controlled rate.
3. **Use a controlled currency set.** A custom config entry, JDK database, dedicated service. Never accept arbitrary currency codes; validate at the boundaries of the system.
4. **Codes identify fiat only.** Currency codes are unique and usable as identifiers only for fiat. For crypto currencies you will have to use a more complicated approach like `(network, contract address)` or similar.
5. **Currencies carry metadata.** Symbol, precision, name, etc. You will usually need those details for display purposes but rarely for business logic.
6. **Pegged is not the underlying.** Pegged, bridged, and wrapped crypto currencies are not equivalent to the underlying ones.

Principles touched:

- **No trust.** Validate currency against the controlled set at the boundary.
- **No invented data.** Treating distinct currencies/assets as interchangeable conjures value.

FX Rates

FX (Forex, foreign exchange currency market) rates allow us to convert money between currencies.

1. **A rate is always directional.** The EUR/USD rate is not the same thing as the inverted USD/EUR rate. On an exchange, buying and selling are two different orders at different prices (the bid/ask spread), so the two directions don't simply invert.
2. **The time of the rate is critical.** While you can technically use a rate from any point in time, the most commonly used are:
   - **Current-time rate.** Used to calculate current holdings or the value of a transaction as if it happened right now.
   - **Value-date rate.** Used to calculate change in value or a tax amount.
3. **Two kinds of rate matter for conversion:**

    1. **Transactional rate.** The rate a real conversion happened at. You don't store it directly - it falls out of the original and result amounts.
    2. **Reference rate** (mid-market or central bank). One used for valuation and equivalence (what holdings are worth right now, or a tax base at the value date) and not a price anyone actually trades at.
4. **There is no canonical rate.** Rates come from markets and vary between venues or calculation methods. The closest to canonical are central bank rates, which can be used only as a reference rate, and even there we can have alternative sources which are just as valid.

Principles touched:

- **No lost data.** Keep the amounts (and, for reference rates, a way back to the source).
- **No trust.** There's no canonical rate, so the source should be part of the data.

Recording money: the ledger

Once represented, money movements have to be recorded in a way that balances, survives audit, and can be reconstructed years later. This is where the books, their timestamps, and their history live.

Double-entry bookkeeping

Double-entry bookkeeping is a widely used way to store financial transactions as a list of entries in the form of `(credit account, debit account, amount)` (this is a compact form; the classic representation uses a separate debit and credit row per movement). Because every entry moves the same amount out of one account and into another, the books always balance - money is only moved, never created or destroyed.

- **Money always has a source and a destination.** External providers get dedicated accounts too, so money entering/leaving the system is still tracked.
- **Balance is never stored.** It's derived from the movements of money.
- **Accounts have a type.** Assets, liabilities, or equity, so the **accounting equation** (`assets = liabilities + equity`) holds and each account has a defined side on which it increases. In practice you also need income (revenue) and expense accounts - e.g. to book a fee as revenue or a write-off as a loss (`assets = liabilities + equity + revenue - expenses`).
- **One transaction, many movements.** A single transaction will usually create multiple movements, e.g. one for the net amount, another for the fees.
- **Posted entries are immutable.** By convention, corrections are made by adding new compensating entries that offset the original.

Principles touched:

- **No invented data.** Money only ever moves between accounts; the total is conserved.

Value time vs booking time vs settlement time

Transactions will usually have at least two, sometimes three timestamps associated:

1. **Value time.** When the transaction occurred.
2. **Booking time.** When the transaction was recorded in the system.

3. **Settlement time.** When money was actually transferred or materialized. Not every transaction has one. Usually expressed as T+X, where X is the number of days after value at which settlement happens (e.g. T+2 means 2 days after value).

The first two will almost always diverge:

- **Backdated** (booking > value). Technically almost all transactions are backdated, but the term is most impactful when booking and value time fall under different reporting periods, e.g. days, months, years.
- **Forward-dated** (booking < value). Less frequent, but happens e.g. with scheduled or future-dated payments - a standing order recorded today but effective next week.

Example: a card payment happened at T1 (value time), you recorded it at T2 (booking time), but the payment provider transferred money to your account at T3 (settlement time).

Business and business-consumed reports usually care about value or settlement time, while booking time is useful for traceability.

Principles touched:

- **No lost data.** Record every relevant timestamp; collapsing them into a single `created_at` loses information you can't reconstruct later.

Audits and audit trails

Financial systems are subject to regulatory scrutiny in the form of various audits. Some of the things that might be verified during an audit:

- are company funds not commingled with user funds or used for company expenses?
- are all revenues registered, reported, and explainable? E.g. can you pinpoint the transactions that contributed to a particular revenue stream in a particular period?
- is the information provided to the external world (e.g. users or the tax office) matching reality? E.g. does the company hold as much in assets as it owes its users?
- are the funds protected against external threats? (e.g. who can access the funds and how)

To answer those and many other questions, financial systems have to keep track of not only the current state but the full history of how that state came to be. This history is the **audit trail**: a record of everything that happened, detailed enough that any balance, report, or decision can be explained and reproduced from it.

A useful audit trail captures, for every change:

- **What** happened.
- **When** it happened (see value time vs booking time).
- **Who or what** triggered it - a user, an operator, an automated job.
- **Why** it happened - a reference to the order, instruction, or incident that caused it.

Money movements are the obvious subject, but manual interventions, configuration changes (fee schedules, rate sources, limits), and permission changes need trails too.

The **why** is often itself the output of a decision (e.g. a compliance check or risk score). Recording just the outcome ("blocked") rarely satisfies an audit because you'll be asked how that outcome was reached. If that logic lives in a decision table or a rules engine (DMN, Drools,

Decisions4s) instead of being buried in imperative code, the decision becomes a structured, replayable artifact that says which rules fired, on which inputs, with what result.

Principles touched:

- **No lost data.** Current state alone can't answer an audit's questions; only the full history can.

Event sourcing

Event sourcing is probably the most principled and systemic approach to building an audit trail. In ES, instead of storing current state with a log next to it, you store only the events and derive state from them. The double-entry ledger is an example of this pattern applied to money - balance is never stored, it is calculated from the stored entries. With this approach the trail is a primary artifact and cannot drift away from reality.

A few practical notes:

1. **You don't need it everywhere.** The ledger already covers money; for surrounding domains a conventional model with a reliable change log may be enough.
2. **Derived state can be cached.** Balances and projections can be cached or snapshotted for performance.
3. **Projections are work intensive.** You might need a lot of them, and you cannot effectively query your primary data set (events) for anything, so you need to build dedicated or generic projections to look into your data.
4. **Plan for schema evolution.** Events live for years, so today's code must still read events written long ago.

In other words: event sourcing is a very good solution when an audit trail is required, but it comes with a very high price in terms of system complexity.

Principles touched:

- **No lost data.** When state is derived from events, the trail can't drift out of sync with reality because it *is* the source of truth.

Immutability

An audit trail that can be edited proves nothing, hence records can never be updated or deleted. Our log must be append-only, and every correction should be a new record (see below).

Immutability is an invariant, and the usual toolbox applies:

1. **By construction.** Append-only tables, revoking `UPDATE`/`DELETE` at the database-permission level.
2. **Runtime checks.** The application layer exposes no mutating operations on posted records.
3. **Post-factum.** Tamper evidence: checksums or hash chains over the records, periodically verified, so that any after-the-fact modification is detectable.

When building a real system bugs are unavoidable and might require you to fix the event log/audit trail. In those cases it's sometimes easier to update the trail in place instead of keeping it strictly immutable. To balance those two worlds it's important to understand your reporting schedule and obligations - usually data has to be kept in stone only once it has been reported, e.g. when the financial statement has been shared at the end of the month. Until then you

might still be able to modify your data in place, if you detect the problem and fix it before it leaves your system.

Principles touched:

- **No trust.** An editable history proves nothing; immutability and tamper evidence make the trail trustworthy to an outsider, including yourself investigating an incident.

Reversals and corrections

Mistakes still happen, for example a wrong amount gets posted or a transaction lands on the wrong account. Immutability means fixing forward - post a new compensating entry and link it to the record it corrects, in both directions.

1. **Reversal.** Negates the original in full, as if it never happened economically - but it stays visible in the history, together with the original.
2. **Correction (adjustment).** Books the difference between what was recorded and what should have been, or reverses and re-posts with the right values.
3. **Mind the reporting period.** Corrections often land in a different reporting period than the original (see value time vs booking time); the linkage is what lets reports attribute them correctly and distinguish real activity from cleanup.

The last point is particularly important - when posting corrections/reversals you will need to decide whether to backdate the event (specify a value time in the past) or not. Here a lot depends on the reporting schedule again - usually you won't be allowed to backdate anything to an already closed period, because it was already reported to the external world.

Principles touched:

- **No invented data.** Mistakes are fixed by posting linked compensating entries that offset the original record.

Immutability vs GDPR

GDPR's right to erasure appears to contradict an immutable ledger. In practice it's quite easy to make it a non-problem:

1. **Financial records are largely exempt.** Legal retention obligations (accounting law, AML, typically 5-10 years) take precedence over erasure requests for transactional data. You don't delete postings in that timeframe.
2. **Separate PII from financial data.** The exemption covers what you are obliged to keep, not everything you'd like to keep. The immutable ledger references users only by opaque internal identifiers, while PII (names, addresses, documents) lives in a separate, mutable store that can be redacted or erased independently.
3. **Crypto-shredding for embedded PII.** Where personal data must be embedded in immutable records (e.g. event payloads), encrypt each user's personal fields with a per-user key and erase by deleting the key. Erasure becomes a key deletion, not a rewrite of history.

Principles touched:

- **No lost data.** Separating PII from financial data lets you honor erasure without losing the financial history you're obliged to keep.

Executing money flows

A money operation is rarely a single write. It spans steps, concurrency, and failure, and has to stay correct - never inventing or losing money - through all of it. These are the patterns that keep a single flow correct, from the invariants it must preserve to surviving a crash mid-way.

Invariants

In any system there exist special properties that must always hold - we call them invariants. One such invariant is the accounting equation mentioned above. Your business stakeholders might define many such conditions that then have to be enforced.

There are 3 primary ways to enforce invariants:

1. **By construction.** Make sure that the system allows creating only valid objects, so invalid states are unrepresentable. This can be done through a variety of techniques: factory methods (smart constructors), type-level programming (e.g. refined types), database constraints.
2. **Runtime checks.** Check that invariants hold when executing logic. This can be assertions in production code or tests - property-based testing shines here (e.g. "for any sequence of postings, the books balance").
3. **Post-factum.** Analyse the data persisted by the system looking for any violations, e.g. reconciliation jobs or nightly checks that ledger balances still satisfy the accounting equation.

What's important: those methods are complementary and you will usually use all of them side by side to achieve the desired level of trust. By construction is the strongest but cannot express everything (especially cross-aggregate or cross-system invariants), runtime checks catch violations at the point of occurrence, and post-factum is the only one that catches bugs that already shipped - but catches them late.

Principles touched:

- **No trust.** Invariants are verified, not assumed; even your own code's output gets checked.

Funds reservation

In most cases your transactions will require interaction with the external world. For example, you might need to run compliance checks before allowing a user to withdraw funds, or you need to register the withdrawal in an external system.

In such cases you also have to avoid race conditions - spending the same money twice, or discovering "insufficient balance" only after the external world interaction already happened.

To address this, systems implement funds reservation (also known as hold-and-release), where funds are first reserved for a particular transaction before the external interaction starts. Once it completes, the reservation is settled and the transaction proceeds; if anything goes wrong, the reservation is released and the funds return to the available balance.

This pattern introduces a distinction between two balances: the **total balance** (everything the user owns, including reserved funds) and the **available balance** (`available = total - reserved`). Balance checks and new reservations are made against the available balance, which is what prevents the same funds from backing two transactions.

A few practical notes:

1. **The final amount may differ.** It's not always known upfront; fees or rates may differ from the estimate. In that case you reserve the estimated amount, settle the actual one, and release the remainder.
2. **A reservation must always resolve.** One that's never settled nor released locks user funds, so every flow that creates one must guarantee it eventually resolves it. An explicit expiry/timeout can serve as a safety net, but it's optional - you can rely on internal system discipline instead. Notably, the failure mode is conservative: an orphaned reservation locks money, it never loses or creates it.
3. **It needs strong consistency.** Checking the balance and recording the reservation must be linearizable. On a stale read, two transactions can both pass the check and back their spends with the same funds. So no eventual consistency here, sorry.

Principles touched:

- **No invented data.** The same funds can never back two transactions; a reservation makes this explicit instead of relying on a racy balance check.

Handling overdrafts

An overdraft happens when an account balance goes negative. Overdrafts come in two kinds:

1. **Intentional.** An overdraft is a credit product the business explicitly offers, with limits and interest. This is a business feature, not an anomaly, and is mostly out of scope here. This will most likely be modeled as a separate overdraft account (liability for the user, receivable for the operator) with a positive balance.
2. **Unintentional.** The balance goes negative even though policy forbids it.

Unintentional overdrafts happen even in correct systems, because the external world doesn't ask for permission: a settlement comes in higher than the reserved estimate or a reversal lands after the funds already left. Funds reservation reduces the window for overdrafts but cannot eliminate it.

Forbidden is not the same as unrepresentable. It's tempting to encode "balance is never negative" at the type or storage level as an unsigned integer or a `CHECK (balance >= 0)` constraint. But when we are forced to accept a negative balance, a system that cannot represent it will either crash mid-flow, silently clamp the balance to zero (inventing money), or do something similarly wrong.

Put differently, "balance >= 0" is just an invariant and the usual toolbox applies: enforce it at runtime when authorizing transactions, detect violations post-factum with monitoring and reconciliation - but don't force it by construction. When an overdraft is detected, it's a signal to investigate but not necessarily a bug.

When an overdraft does happen, we have to book it and recover explicitly, e.g. by netting it against future deposits, requesting repayment, or writing it off - as an explicit compensating entry to an expense/loss account.

Principles touched:

- **No invented data.** Clamping a negative balance to zero mints money.
- **No trust.** The external world can force an overdraft no matter what your checks say.

Idempotency

In a distributed system it's impossible to guarantee exactly-once delivery - any call can be interrupted and we won't know whether it reached the other side or not. To make sure a message is delivered, we have to retry every such call. But in doing so we risk delivering it more than once, hence its processing needs to be idempotent - the same message delivered twice must trigger the processing only once.

1. **Prefer explicit keys.** Idempotency keys vs business-derived idempotency (e.g. deduplicating on the payload). An explicit key is usually the simpler and better solution - deriving it from the data is fragile, e.g. it's hard to tell whether two transactions with the same amount are a duplicate or two genuine operations. When using idempotency keys make sure they are scoped to a particular operation and client.
2. **Decide how errors replay.** When a call failed the first time, should a retry re-raise the stored error or re-trigger the processing? It's usually simpler and easier to reason about when we treat the error as the idempotent result and replay it. The client can always retry with a new key. A lot depends on the nature of errors - permanent ones (e.g. validation) should be replayed as-is while temporary ones (e.g. network failure) might be reprocessed.
3. **Validating the repeated payload.** It's good practice to ensure a repeated call carries the same payload as the original. In practice this gets costly and buys only a little extra confidence, at the cost of a more complex implementation and less flexibility (the caller might change the request for a good reason).
4. **At scale it's hard.** Building reliable idempotency can be a complex endeavour, so dedicate enough effort to it. Not only might you need to deduplicate billions of requests, but you also have to get the behavior right under concurrent access (e.g. two duplicate calls arriving in the same millisecond). Your idempotency barrier has to be atomic.
5. **Beware time windows.** You might be tempted to rely on an idempotency time window, e.g. dedupe only within 24h. This significantly simplifies the implementation (otherwise the data volume grows forever) but at the cost of correctness. Make this tradeoff only if you absolutely have to, because it will haunt you later.
6. **Test for retries.** One of the better approaches is to bake a generic middleware into your integration or system tests that automatically repeats every call.
7. **Handle out-of-order retries.** Your system needs to stay idempotent even if it already moved to a new state - e.g. keep putting the funds on hold idempotent even if they were already released.

Idempotency matters on both sides - when you make calls and when you receive them. Keep it in mind every time you consume or expose an operation.

Principles touched:

- **No invented data.** Retries are unavoidable, so processing must collapse duplicate deliveries into a single effect instead of moving money twice.

Full resumability

A money flow rarely happens in a single step. A withdrawal might reserve funds, run compliance checks, register the operation in an external system, and finally settle. Such a sequence is stretched across time and can die between any two of those steps, so the safe assumption is that it will: assume failure every two steps. A flow can therefore never assume it runs to

completion in one go, and a half-finished one must always land in a recoverable state, never an inconsistent one.

1. **Persist progress, don't keep it in memory.** Model the flow as an explicit state machine whose state is durably stored, and commit each step's completion before starting the next. A restart must be able to tell exactly where the flow was.
2. **Something must resume stalled flows.** An independent driver (a scheduler, worker, or poller) has to pick up incomplete flows and push them forward. A crash of the orchestrator must not strand a flow forever.
3. **Every step must be safe to re-run.** On resume you may re-execute a step that already partially happened, so each one has to be idempotent (see idempotency).
4. **Roll forward or compensate.** External effects can't be rolled back. Once you've called the outside world you can't un-call it, and a database rollback won't undo it. So you either retry forward until the flow completes, or, when a later step fails for good, post compensating actions to unwind the earlier ones (the saga pattern).

You can use a durable-execution engine (e.g. Temporal, Camunda, Workflows4s, AWS Step Functions) or hand-roll your own persistent state machine.

Principles touched:

- **No lost data.** A crash in the middle of a flow must never lose track of in-flight money; persisted progress is what lets the flow be picked up and finished.
- **No invented data.** Resuming re-runs steps, so they must re-apply without double-counting - the flow completes exactly once.

The external world

Interacting with the external world - whether in the form of 3rd party providers (payments, KYC, AML, banks, custodians, etc.) or internal services - is unavoidable. Our job is to build a system that stays correct regardless of how unreliable those dependencies become.

Consuming APIs

Sooner or later you will have to call someone else's API, e.g. a payment provider, a custodian, a blockchain node, or a KYC vendor. You don't control its code, its quality, or its uptime, so the safe default is to assume it will misbehave and to build defensively around it.

1. **Don't trust the schema.** The response will not always match the contract you were given: fields can go missing, types can change, nulls can appear where they shouldn't. Validate the important pieces at the boundary and fail loudly on anything you didn't expect, so malformed data cannot leak into the system. At the same time, never validate the pieces you don't need, as it might cause unnecessary outages when a third party breaks its contract. And they will.
2. **Expect imperfect engineering.** Everything you consider a questionable engineering practice will rear its head given enough time: tokens passed in URLs, lost precision, HTTP codes that don't mean what they should (a `200` carrying an error body), inconsistent pagination, custom date formats. Don't get frustrated by it; treat it as the job rather than the exception.
3. **All calls will fail.** Design the system so that it can handle a lack of response. Retries and timeouts are necessary protection.

4. **Circuit breakers are usually optional.** They are mostly a courtesy toward an overloaded server, paid for by you with added complexity on the client side. It's reasonable to expect the server to handle its own load and drop requests it can't serve. That being said, a circuit breaker also protects your latency and finite resources (threads, connections, etc.), so employ one when it's really needed.
5. **Mind the quotas.** Rate limits and usage quotas are easy to forget but can be a source of nasty weekend outages. It's good to do a bit of napkin math up front (expected call volume against the provider's limits) so you find out before it causes a problem.
6. **Store every request and response.** It might sound excessive, but it can be a lifesaver during an investigation when an external API starts returning something it never should. Persist what you sent and what came back, in a structured, queryable form (e.g. a Redshift table). This will also be your audit trail and evidence when the provider's behavior is disputed, and your material for reprocessing after a bug.
7. **Aim for provider redundancy.** For the most critical parts, consider using more than one provider for the same purpose. You can never fully trust the provider, so when the stakes are highest this can mean validating the data against multiple sources (e.g. two blockchain nodes) or having a backup bank partner, crypto custodian, or KYC vendor. This approach is extremely expensive (development, fees, and complexity-wise) but might be necessary to achieve the desired level of reliability.
8. **Don't trust the sandbox.** If a provider gives you testing/sandbox access, that's already a good sign. Those environments are fine for basic scenarios but will usually diverge very significantly from the production setup. Be prepared to test in production (e.g. through canary releases and controlled usage with small impact).

Principles touched:

- **No trust.** The provider's code, schema, and uptime are all outside your control, so verify facts against independent sources and validate everything at the boundary.
- **No lost data.** Persisting every request and response keeps a record you can reconcile against and reprocess from.

Handling webhooks

Webhooks are the most common way to receive signals from external systems, but processing them safely is not trivial. While we focus here on webhooks (HTTP endpoints you expose, called by an external system with a payload defined by that system) many of the points apply to other transport methods as well.

1. **Don't assume ordering.** Messages can arrive out of order or carry stale data, so the last webhook you received is not necessarily the latest truth. Don't blindly overwrite your state with whatever just arrived; reconcile it against what you already know (e.g. by querying the API for the current state).
2. **Don't assume validity.** Webhooks might come from a secondary part of the issuer's system and carry stale or improperly transformed data. A good practice is to ignore the content of the webhook and use it only as a trigger to query the API for the authoritative state. Beware that the API can be eventually consistent and lag behind the webhook, so a query right after the trigger may still return the old state - be ready to retry.
3. **Don't assume delivery.** Webhooks will get lost sooner or later, regardless of how strong a re-delivery policy the issuer promises. You have to be prepared to handle a missing

webhook, which usually means an independent process that fixes the completeness of your data. See reconciliation.

4. **Don't assume single delivery.** The same webhook will be delivered more than once. Processing must be idempotent. See idempotency.
5. **Acknowledge fast, process asynchronously.** Return a 2xx as soon as you've durably stored the raw event, and do the real work asynchronously. If you process inline and are slow, the issuer can time out and retry, multiplying your load.
6. **Persist the raw payload.** Store what you received verbatim before acting on it. It will not only make processing more reliable but will also act as your audit trail of what the provider actually said. It also lets you reprocess the message after a bug without asking the provider to resend.
7. **Verify the caller.** The usual mechanism is for the issuer to attach a signature of the payload, so you can verify the message really came from them. Most commonly this is an HMAC computed with a shared secret; less commonly it's an asymmetric signature whose public half is published. One caveat: verify the signature over the *raw bytes* you received, not a re-serialized payload (re-serialization changes bytes and breaks the signature). Even with this, prefer not to trust the content (see point 2).

There is a recurring theme here: don't trust the webhook. Treat it as a hint that *something* happened, not a trustworthy account of *what* happened.

Principles touched:

- **No trust.** A webhook is an unauthenticated, unordered, possibly-lost, possibly-duplicated hint; verify the source and confirm the actual state against the API.
- **No lost data.** Persist the raw event and back delivery up with reconciliation so a dropped webhook doesn't mean a dropped fact.

Notifying reliably (Outbox and CDC)

It's quite often a requirement to let the external world know about changes in our system in a reliable way - by publishing a Kafka event, dispatching a webhook call, or through a plethora of other means. The problematic part is *reliably*: we have to ensure at-least-once delivery, and those channels don't fit the usual transactionality model we tend to rely on. Without transactionality we risk either:

- **Publish then rollback.** The publish succeeded but we didn't get the response due to a network issue, so we roll back our system's state.
- **State change without publish.** The publish genuinely failed but we didn't roll back.

The textbook answer is a 2-phase commit/distributed transaction, but it's rarely used due to its complexity and the lack of a good way to standardize and reuse the approach. The practical options:

1. **Outbox pattern.** A "publishing" event is written transactionally (with the state change) into a dedicated store, and from there it's reliably processed (take a row, retry until success). In other words, we reliably save "publishing intent" and then process it later.
2. **Change Data Capture (CDC).** An automated mechanism that detects changes committed to the database (typically by tailing its write-ahead/replication log) and turns them into a stream of events. Because it reads straight from the log, every committed change is captured and nothing is missed, without any explicit publishing code in the application.

Tools like Debezium or AWS DMS implement this off the shelf. The tradeoff is coupling and operational weight: raw CDC emits events shaped like your table rows and needs postprocessing to avoid leaking the internal schema to consumers.

3. **Listen-to-yourself.** Reverse the order and publish the event first (e.g. to Kafka), then rebuild our own state from it.
4. **Event sourcing.** The event log already lives in the database, so publishing is just a matter of reading from it (see Event sourcing).

Whichever mechanism you pick, delivery is at-least-once - the relay or connector can publish and then crash before recording that it did, re-sending on restart. Consumers must therefore be idempotent and deduplicate on a stable event id (see idempotency).

Principles touched:

- **No lost data.** A committed change must reliably reach its consumers; the outbox (or the log) guarantees the notification can't be dropped just because a separate publish step failed.
- **No invented data.** We never publish a notification for a change that didn't commit, and duplicate deliveries collapse into a single effect.

Reconciliation

Any system that relies on external data is prone to data drift - a situation where one system doesn't match the other. For example, you might miss a webhook, or a transaction might be posted to the ledger but not reflected in the external provider's system. In all such cases we need reconciliation: a process that aligns the two systems. While we say "two", in practice it can be more than that, e.g. ledger, payment processor, and the bank, but this doesn't change anything in how to approach the problem.

1. **Cadence.** Depending on the exact context and constraints, reconciliation might be done hourly, daily, monthly, or even yearly.
2. **Nature of drift.** Data can be missing (which is an easy case) or different (e.g. the same transaction with different amounts, which is much more complicated to solve). Timing also matters a lot: if settlements happen at T+3, records will stay unreconciled for 3 days - that logic should be incorporated into the process so that we don't alert on those cases.
3. **Matching algorithm.** Knowing what to compare between the two systems is the hard part. Usually you want to persist the external provider id within your system so that matching is straightforward. If this is not the case, heuristic algorithms enter the game (e.g. matching by amount and time).
4. **One-to-many.** In some cases you will have to reconcile multiple records on one side with one on the other, e.g. a single settlement transfer might cover a couple of transactions.
5. **Aligning is not trivial.** It goes without saying we can't simply overwrite the data to make the reconciliation happy. Each discrepancy found should be understood and fixed through first-class support, e.g. a correction record, reprocessing of webhook data etc.

Principles touched:

- **No trust.** Reconciliation is how we verify across independent sources instead of believing any single one is right.
- **No lost data.** It's the safety net that catches the dropped fact - the missing webhook, the unsettled transfer - before it disappears for good.

Controls and access

The patterns so far keep the *data* correct. But a money system also has to constrain who is allowed to act on it, and prove after the fact that the process was followed. This is where the **No trust** principle turns inward and your own operators and engineers are a trust boundary too, just like external providers and internal components. An auditor examines these controls alongside the books themselves.

Segregation of duties and four-eyes

Some actions are too sensitive to leave to a single person, regardless of how trusted they are. Splitting them is the oldest control in finance, and it takes two related forms: **segregation of duties** (no one person owns a whole process) and four-eyes / maker-checker** (a specific action needs a second person to approve it before it takes effect, also called dual control).

1. **It applies to money operations.** Large or manual withdrawals, manual ledger corrections, treasury and cold-wallet moves, changing a fee schedule or a limit - anything that can move or misstate funds is a candidate for a second approver.
2. **It applies to engineering too.** Merging code, deploying to production, and changing infrastructure are sensitive actions in a money system. Hence we usually require review and approval.
3. **The approval is part of the trail.** Record who requested, who approved, and that the two were different people - otherwise the control is unprovable (see Audits and audit trails).
4. **Break-glass needs a path.** Emergencies happen, and a rigid control invites people to route around it. Provide an explicit, heavily-audited override rather than forcing a backdoor.

Principles touched:

- **No trust.** A single internal actor - even a trusted one - is not sufficient authority for a sensitive or irreversible action.

Access control

Who can do what is itself part of the system's state, and it changes over time as people join, move teams, and leave. It's not enough to know who can touch funds today; Auditors will also ask how they came to have that access.

1. **Least privilege.** Grant each actor - human or service - the minimum needed, and prefer roles (RBAC) over per-person grants so that access stays reviewable.
2. **Authorization changes need a trail.** Granting or revoking a capability is a sensitive event, exactly like a money movement: record what changed, who changed it, and why. The audit-trail discipline from the ledger applies here too (see Audits and audit trails).
3. **Review access periodically.** Permissions become stale or inaccurate. Scheduled access reviews (recertification) are the post-factum check (see Invariants) applied to access so that we can catch the drift.

Principles touched:

- **No trust.** Standing access quietly accumulates; least privilege and periodic review are what keep it in check.

The change trail (SDLC)

In a regulated environment we usually have to audit how code reaches production and so know who reviewed a change, who approved it, when it shipped, and so on. Your version control and CI/CD systems are a great help here if done right.

1. **Source control is the record.** Commit history attributes every change to an author and ties it - through review and linked tickets - to the reason it was made (it's the usual *what / who / why* an audit trail demands). Protect it accordingly via signed commits, protected branches, no force-pushing shared history.
2. **Reviews and pipelines must be enforced.** Required (non-optional) reviews, status checks, and "no direct pushes to main" are crucial because discipline doesn't fly in audits.
3. **Deployments are traceable.** Which version is running, who released it, and when, should be reconstructable - this is what lets an incident be tied back to the change that caused it.

Principles touched:

- **No lost data.** The history of how the system itself came to be is as much a part of the trail as the history of the money it holds.
- **No trust.** The system enforces the delivery controls - it doesn't rely on people remembering to follow them.

Testing

Tests matter everywhere, but in a money system they matter more. The difficulty is that you usually can't enumerate the expected outputs - the space of operation sequences is too large and the interesting failures live in the combinations. The approaches below are ways to gain confidence in the correctness of our system. Treat it as a restaurant menu from which you can choose the techniques with the most impact on your system.

1. **Property-based testing.** Instead of asserting specific outputs, you assert that a property holds for any generated input. This is a natural fit for invariants or money math. The framework generates the awkward cases you wouldn't think to write by hand.
2. **Invariant checks between steps.** When you generate a sequence of operations, don't only assert the invariants at the end - assert them after every single step. This is impossible to do manually at scale, so you would need a more sophisticated testing harness that automatically injects the assertions.
3. **Generative idempotency testing.** Since every operation that touches the outside world has to be idempotent (see idempotency), you can make that a property of your system. Using a similar approach to the one above, you can automatically repeat all the declared operations and assert the lack of impact on the system from the second call.
4. **Crash and resume injection.** Long flows must survive dying between any two steps (see full resumability), and we can test exactly that by following the usual approach: inject a failure at every step.
5. **Round-trip testing.** Encode then decode, serialize then deserialize, convert then convert back - and assert you land where you started (or within a known tolerance). It's a quick way to catch precision loss at boundaries and serialization bugs in your money and currency types. It plays really well with automatic data generation.
6. **Golden testing.** Pin the output of a calculation or projection (a fee breakdown, a statement, a report) to a stored expected result, so any unintended change shows up as a diff. Useful

for the gnarly, hard-to-reason-about computations where you trust a reviewed-once result more than a freshly written assertion.

7. **Backward-compatibility testing.** Events and stored records live for years, and today's code must still read what old code wrote (see event sourcing). Keep a corpus of real, old-format payloads and assert that current code still deserializes and projects them correctly - this is what stops a schema change from silently breaking history.
8. **Testing in production.** Some confidence is only obtainable against the real thing. Provider sandboxes diverge significantly from production (see consuming APIs), so the final proof that an integration works often has to happen live - through a canary release, a controlled rollout with a small blast radius, or synthetic transactions that push small real amounts through the system continuously as a health check. The money-specific caveat is that these are real movements: a test in production moves real money, so it must go through the same ledger, reconciliation, and audit trail as everything else, be clearly tagged, and be cleaned up through the normal correction/reversal machinery - never a backdoor that bypasses the books.

Principles touched:

- **No trust.** Tests are how you verify the patterns actually hold rather than assuming they do; the invariant is the oracle, not a value you happened to expect.
- **No invented data.** Replaying operations and injecting failures proves that retries and recovery don't double-count or mint money.
- **No lost data.** Round-trip and backward-compatibility tests prove that precision and history survive boundaries and the passage of time.

Appendix A: Know your domain

The hardest part of joining fintech is often not the code but the vocabulary and concepts behind it. The field is full of words that sound ordinary but mean something precise, and acronyms that everyone around you uses without ever expanding.

Caveats: terms a layperson already knows (deposit, withdrawal, transfer, currency) are skipped, and so are the exotic corners you can learn when you get there. We try to focus on the most important terms. Where the handbook already covers a concept properly, the entry links to that section instead of repeating it.

Accounting & ledgers

- **Ledger** - the system of record for money movements; the source of truth from which balances are derived (see Double-entry bookkeeping).
- **General ledger vs sub-ledger** - the single consolidated book vs a detailed book for one domain (e.g. one per user or product) that rolls up into it.
- **Debit / credit** - the two sides of every entry. Which one *increases* an account depends on the account's type, not on "money in vs money out" (see Double-entry bookkeeping).
- **Posting** - committing an entry to the ledger; "posted" means recorded and, by convention, immutable.
- **Chart of accounts** - a catalogue of the accounts that can be posted to; a single system can have several (e.g. per legal entity, per book, or per reporting standard).

- **Account type** - asset / liability / equity (plus revenue / expense), so the **accounting equation** holds and each account has a defined side on which it increases (see Double-entry bookkeeping).
- **Receivable / payable** - money owed *to* you / money owed *by* you.
- **IOU** - informally, a liability: a record that you owe someone money. A user's balance on a custodial platform is an IOU from the platform to the user, which is why it sits on the liability side of the books.
- **Accrual vs cash basis** - recognising money when it's earned or owed vs when it actually moves.
- **Trial balance** - a check that total debits equal total credits across the books.
- **Suspense / clearing account** - a temporary holding account for money that's in transit or not yet attributable.
- **Write-off** - booking a balance you no longer expect to recover as a loss (see Handling overdrafts).
- **Commingling** - mixing company funds with user funds; a regulatory red flag (see Audits and audit trails).
- **Reconciliation break** - a single unmatched discrepancy surfaced by reconciliation.

Money & FX

- **Money (as a type)** - an amount paired with a currency (see Currency handling).
- **Minor units** - the smallest indivisible unit of a currency; amounts are often stored as integers of these (€12.34 ⟶ `1234`) (see Precision handling).
- **Basis point (bp / "bip")** - one hundredth of a percent (0.01%); fees and rates are routinely quoted in these.
- **Notional** - the face value a calculation is based on, which may be far larger than the cash that actually changes hands.
- **Fiat vs crypto** - state-issued currency vs blockchain-native asset
- **Stablecoin** - a token pegged to a reference asset, usually a fiat currency like USD.
- **Pegged / wrapped / bridged** - representations connected to an underlying asset but *not* equivalent to it (see Currency handling).
- **Bid / ask / spread** - the buy price, the sell price, and the gap between them.
- **Mid-market rate** - the midpoint between bid and ask; a reference point, not a price you can actually trade at (see FX Rates).
- **Reference rate** - a rate used for valuation and equivalence (holdings value, a tax base), not for an actual trade (see FX Rates).
- **Mark-to-market** - revaluing a holding at the current market rate rather than the price it was acquired at.

Transactions, timing & settlement

- **Value date / booking date / settlement date** - when it happened / when we recorded it / when money actually moved (see Value time vs booking time vs settlement time).
- **T+X** - something (e.g. settlement) happens X business days after the value date (e.g. T+2).
- **Clearing vs settlement** - agreeing who owes what vs actually transferring the money.
- **Cut-off time** - the daily deadline after which a transaction rolls into the next settlement window.
- **Float** - money that, mid-transfer, appears to exist in both systems at once (or in neither).

- **Netting** - offsetting many obligations into a single net transfer instead of settling each one gross.
- **Backdating** - assigning a value date earlier than the booking date (see Value time vs booking time vs settlement time).
- **Reversal / correction** - negating a posting in full as if it never happened economically vs booking the difference between what was recorded and what should have been (see Reversals and corrections).

Payments, rails & cards

- **Payment rail** - the underlying network a payment travels over (SEPA, SWIFT, ACH, card networks, a blockchain).
- **IBAN / SWIFT / SEPA / ACH / FPS / CHAPS / wire** - identifiers and networks for moving fiat between banks.
- **Originator / beneficiary** - who sends / who receives a transfer (also remitter / payee, payer / payee).
- **PSP (payment service provider)** - a vendor that connects you to one or more rails.
- **Nostro / vostro account** - "our money held at their bank" / "their money held at ours"; the accounts that make cross-bank transfers work.
- **Omnibus account** - one pooled account holding many users' funds together, with per-user balances tracked internally. Legitimate pooling, as opposed to commingling.
- **FBO ("for benefit of") account** - an account a company holds on behalf of its users.
- **Sweep** - automatically moving balances between accounts on a schedule (e.g. into cold storage or an interest-bearing account).
- **Chargeback** - a forced reversal of a card payment, initiated by the cardholder's bank.
- **Issuer / acquirer** - the cardholder's bank / the merchant's bank.
- **Interchange** - the fee the acquirer pays the issuer on each card transaction; the bulk of card-processing cost.
- **Authorization vs capture** - placing a hold on funds vs actually charging them; the card world's version of funds reservation.
- **Dunning** - the retry-and-notify process for recovering a failed recurring payment.

Trading & markets

- **Order book** - the live list of outstanding buy (bid) and sell (ask) orders at each price level.
- **Market vs limit order** - execute now at the best available price (takes liquidity) vs only at a set price or better (rests on the book, provides liquidity).
- **Maker / taker** - maker adds a resting order to the book, taker crosses the spread and removes one; fees usually differ.
- **Slippage** - the difference between the expected price and the price actually filled.
- **Liquidity / depth** - how much can be traded before the price moves; thin books move more.
- **Spot** - buying or selling the asset itself for immediate delivery.
- **Derivative** - a contract whose value derives from an underlying asset rather than the asset itself.
- **Futures / perpetual (perp)** - a contract to trade at a future date / one with no expiry, kept near spot by a funding rate.
- **Funding rate** - periodic payment between longs and shorts that tethers a perp's price to the index.

- **Long / short** - a position that profits when the price rises / falls.
- **Leverage / margin** - controlling a position larger than your capital / the collateral posted to open and maintain it.
- **Liquidation** - forced closing of a position when its margin falls below the maintenance requirement.
- **Haircut** - a discount applied to the value of collateral to cushion against price moves.
- **Counterparty** - the other side of a trade or contract; "counterparty risk" is the risk they fail to deliver.
- **AUM / AUC** - assets under management (client assets the platform actively manages, usually under a fee-bearing mandate) vs assets under custody (client assets it merely safekeeps). The same holdings can be one, the other, or both.

Custody & crypto

- **Custody** - who controls the assets: self-custody (the user holds the keys) vs custodial (the platform or a custodian holds them).
- **Hot / cold wallet** - keys kept online for quick access vs kept offline for security.
- **Private key / public key / address** - the secret that authorises spending / its derived public identifier / where funds are received.
- **Seed phrase** - the human-readable backup that reconstructs a wallet's keys.
- **Multisig / MPC** - requiring several keys (or key-shares) to authorise a transfer, so no single device can move funds alone.
- **Gas / network fee** - the fee paid to get a transaction included on a blockchain.
- **Confirmation / finality** - a block built on top of the one containing your transaction / the point at which it can no longer be reversed. More confirmations = more finality.
- **Reorg** - a blockchain reorganisation that can undo recently confirmed transactions; the reason finality isn't instant.
- **Mempool** - the pool of pending transactions waiting to be included in a block.
- **UTXO vs account model** - Bitcoin-style "spend whole coins, receive change" vs Ethereum-style running balances; the two demand different accounting.
- **Token vs coin** - an asset issued on top of a chain (e.g. an ERC-20) vs a chain's native asset (BTC, ETH).
- **Dust** - an amount so small that the network fee to move it exceeds its value.
- **Address whitelisting (allow-listing)** - restricting withdrawals to a set of pre-approved addresses.

Compliance & regulation

- **KYC** - Know Your Customer; verifying a user's identity.
- **AML / CFT** - Anti-Money-Laundering / Countering the Financing of Terrorism; controls to detect and prevent illicit funds.
- **Sanctions screening** - checking parties against sanctioned-entity lists.
- **PEP** - politically exposed person; a higher-risk customer category requiring extra diligence.
- **SoF / SoW** - source of funds / source of wealth; evidence of where a customer's money came from.
- **Travel Rule** - the requirement to share originator and beneficiary information on transfers above a threshold.

- **VASP** - Virtual Asset Service Provider; the regulatory label for a crypto business such as an exchange or custodian.
- **MiCA** - the EU's Markets in Crypto-Assets regulation.
- **Segregation of duties** - splitting a sensitive action across people so no single person can complete it alone (see Segregation of duties and four-eyes).
- **Four-eyes / maker-checker / dual control** - requiring a second person to approve a sensitive action before it takes effect (see Segregation of duties and four-eyes).
- **Least privilege / RBAC** - granting each actor the minimum access needed / managing it through roles rather than per-person grants (see Access control).
- **Change management** - the controlled, traceable process (review, approval, deployment) by which code and config reach production (see The change trail (SDLC)).
- **Audit / audit trail** - external scrutiny that the books and controls reflect reality / the recorded history that lets any balance or decision be reproduced (see Audits and audit trails).

Resources

No single book covers money systems end to end, so the list below is grouped by layer. Each entry notes what it covers and who it's pitched at, so you can pick by what you're missing.

Accounting & ledgers

- *Accounting for Computer Scientists* (essay, free online) - maps double-entry bookkeeping onto a graph/data model, written for engineers rather than accountants.
- *The Accounting Game: Basic Accounting Fresh from the Lemonade Stand* - accounting from first principles via a running lemonade-stand example; assumes no finance background.
- Modern Treasury, *How to Scale a Ledger* (article series, free online) - building a production ledger from a software-engineering angle.

Payments & cards

- *Payments Systems in the U.S.* - a reference-style tour of how money moves between banks (cards, ACH, wires, checks). US-centric; thorough and on the dry side.
- *The Anatomy of the Swipe* - what happens between tapping a card and the money arriving, pitched at builders and beginners.

Markets & trading

- *Trading and Exchanges: Market Microstructure for Practitioners* - where order books, makers/takers, and spreads come from; a long, detailed practitioner reference.

Crypto

- *Mastering Bitcoin* and *Mastering Ethereum* - engineering-level references for the two models a fintech engineer keeps meeting (UTXO and account). Technical; aimed at developers.

The engineering half

- *Designing Data-Intensive Applications* - idempotency, logs, consistency, and the failure modes this handbook keeps returning to, from the systems side.

KYC & AML

These are written for compliance professionals rather than engineers, so reach for them only when you need the domain itself, not the integration.

- *Anti-Money Laundering in a Nutshell* - a short, awareness-level introduction to what laundering is and how detection and reporting work.
- *Mastering Anti-Money Laundering and Counter-Terrorist Financing* - a heavier practitioner guide to building an AML/CTF framework, with checklists and example documents.

Appendix B: End to end examples

The body of this handbook takes each pattern on its own, which makes it hard to build holistic understanding and intuition. This appendix walks through common flows that act as simplified but representative examples of what you will see in a real system. A production implementation would have more steps, more failure branches, and more bookkeeping, but the simplified version should be enough to get the general idea. They cover the three directions money moves: **out** of the system (a withdrawal), **into** it (a card deposit), and **within** it (a conversion).

Flow 1: A crypto withdrawal

A user asks to withdraw 0.5 ETH to an external address. This is the richest of the three because money is leaving the system through an irreversible external effect - once the chain confirms the send, there is no taking it back.

1. **The request arrives with an idempotency key.** The client may retry the submit (the network ate the response, the user double-clicked), so the very first concern is that two deliveries of the same request create *one* withdrawal, not two (see Idempotency). The key is scoped to this user and this operation.
2. **Funds are reserved before anything irreversible happens.** You reserve 0.5 ETH plus an estimated network fee against the user's *available* balance (see Funds reservation). The balance check and the reservation are a single linearizable step - otherwise two concurrent withdrawals could both pass the check and back their spends with the same coins (see Invariants). From here the user's *total* balance still includes the reserved amount, but their *available* balance does not.
3. **The compliance gate runs - and the flow may sleep here for days.** Before broadcasting, you screen the transaction (sanctions, AML, the destination address). Here we tie several patterns together:
   - It is an **external call**, so it will be slow, fail, or lie, and you build defensively around it (see Consuming APIs).
   - It may escalate to **manual review** that takes hours or days, so the flow has to survive dying between this step and the next (see Full resumability): its state is persisted, an independent driver resumes it, and meanwhile the reservation simply stays in place.
   - A **daily withdrawal limit** is enforced here, and it is nothing more than an invariant (see Invariants). Because it is a time-windowed, stateful invariant evaluated under concurrency, it has the same atomicity problem as the reservation: two withdrawals racing past the limit must not both pass.
   - Every decision - passed, blocked, who overrode it - lands in the audit trail (see Audits and audit trails).
4. **The transaction is broadcast on-chain.** Once cleared, you sign and broadcast through a node - another external call subject to all the same caveats. This step must be idempotent: a resume after a crash must *re-check the chain*, not blindly broadcast a second time (see Idempotency, point 7 on out-of-order retries). The real network fee is not known upfront,

which is exactly why you reserved an *estimate* - you will settle the actual amount and release the remainder.

5. **You wait for finality, then post to the ledger.** A single confirmation is not the end - a reorg can undo a send you declared "done" too early, so you wait for enough confirmations. Only then do you post the double-entry movements: debit the user's account, credit the external on-chain account (the outside world gets an account too), and book the network fee against an expense account and your service fee against a revenue account (see Double-entry bookkeeping).
6. **A nightly job reconciles against the chain.** Independently of the flow above, a job compares the ledger with on-chain reality and the node's view of your transactions (see Reconciliation). It is the safety net that catches a broadcast that never confirmed, or a fee that came out different from what you booked.

Where it gets interesting: suppose the actual network fee comes in higher than the estimate you reserved. The settlement can push the account negative. You book the overdraft and recover it explicitly (see Handling overdrafts). And if the process crashes between broadcast (step 4) and confirmation (step 5), resumability plus idempotency are what let it pick up by querying the chain rather than sending the funds twice.

Flow 2: A card deposit

A user tops up their account with a card payment through a payment service provider (PSP). Money is coming *in*, which shifts the hard part from "don't send twice" to "don't trust what the outside world tells you, and don't credit money that hasn't really arrived."

1. **The user initiates the deposit.** They enter an amount and submit their card details, and you open a deposit transaction with the PSP - again with an idempotency key, since the submit can be retried.
2. **Authorization places a hold.** The PSP authorizes the card, placing a hold without yet capturing the money. This is the card world's version of funds reservation (authorization vs capture). You do not credit the user's balance yet - the money is not yours.
3. **A webhook says "captured" - and you believe none of it.** The PSP calls your webhook endpoint (see Handling webhooks). You verify the signature over the *raw bytes*, persist the raw payload, acknowledge fast with a 2xx, and only then process asynchronously. Crucially you treat the webhook as a *hint that something happened*, not as truth: you query the PSP's API for the authoritative state, because webhooks arrive out of order, carry stale data, get duplicated, and get lost. Processing is idempotent (see Idempotency) so a re-delivered "captured" credits the user once.
4. **The credit goes through a clearing account.** The money is in flight (float) - captured by the PSP but not yet settled to your bank - so you post it through a suspense/clearing account rather than pretending it has arrived: credit the user's balance (a liability - their balance is an IOU from you to them) and debit a PSP receivable. The interchange/processing fee is booked as an expense. Booking time is now; settlement time is later, T+X (see Value time vs booking time vs settlement time).
5. **Settlement arrives as a batch, and reconciliation is one-to-many.** Days later the PSP settles a single transfer to your bank covering many deposits at once. You reconcile that batch against the clearing account (see Reconciliation), matching one settlement against many transactions. The T+X delay is baked into the process so you do not alert on trans-

actions that are simply not settled *yet* - and this same job is what catches a webhook that never arrived at all.
6. **Weeks later, a chargeback.** The cardholder disputes the payment and their bank forces a reversal. You do not edit the original posting - you post a linked compensating entry (see Reversals and corrections), which will usually land in a later reporting period than the deposit, and which may push the user's balance negative if they have already spent the funds (back to Handling overdrafts).

Where it gets interesting: the whole flow is built on *not* believing the happy-path signal. The webhook is a trigger, not a fact; the clearing account refuses to recognize money until it has actually moved; reconciliation verifies the PSP against your own books rather than the other way around. Every step is an application of *no trust*.

Flow 3: An in-app conversion with cashback

A user converts 1,000 EUR into USDC and earns a small promotional cashback on the trade. Money moves entirely *within* the system, so there is no unreliable external rail - instead this flow stresses the representation layer (precision, rounding, currencies, rates) and the sharpest form of the *no invented data* principle.

1. **A directional quote, and a reservation.** You quote a rate for EUR→USDC. That rate is its own price, not the inverse of USDC→EUR - buying and selling sit on opposite sides of the bid/ask spread (see FX Rates). You reserve the 1,000 EUR (see Funds reservation), and the request carries an idempotency key like any other.
2. **The two sides never get added together.** EUR and USDC are distinct currencies - in fact USDC is identified by `(network, contract address)`, not a bare code, and is not interchangeable with the fiat it is pegged to (see Currency handling). The system forbids cross-currency arithmetic; the only bridge between the two amounts is an explicit conversion at the controlled rate.
3. **Full precision through the math, rounding only at the edge.** You compute the conversion keeping full precision and round exactly once, at the boundary, with a deliberately chosen strategy (see Precision handling and Rounding strategies). The spread you earn is *revenue* - it must be booked explicitly to a revenue account via double-entry, not allowed to disappear into a rounding residual (see Double-entry bookkeeping). You do not store the transactional rate as a separate field; it falls out of the two amounts. A separate reference rate is what you would use later to value the holding.
4. **The cashback is the hardest test of "no invented data."** It is tempting to treat a bonus as a free balance bump, but that would mint money from nothing. The cashback is real money: it must be *funded* - moved out of a company promotional/expense account into the user's balance via a proper double-entry posting - and the percentage that defines it needs the same explicit rounding decision as everything else (see Rounding strategies).
5. **Settle, then notify reliably.** You settle the reservation, post all the movements with their timestamps and audit trail, and publish the outcome so the rest of the system - statements, notifications, analytics - learns about it. That publish has to be reliable even though it spans a separate channel, which is what the outbox (or CDC, or the event log) is for (see Notifying reliably (Outbox and CDC)); downstream consumers dedupe on a stable event id because delivery is at-least-once.

Where it gets interesting: the cashback and the spread pull in opposite directions on the same posting. The spread is money the user *loses* to you (revenue); the cashback is money you *give* the user (an expense). Both are real, both go through the books, and both round - so the one transaction has to satisfy *no invented data* (the books still balance, nothing minted) and *no lost data* (every residual tracked) at the same time.

Appendix C: About this handbook

Who wrote it

My name is Voytek and I'm the author of this handbook. The content is based on my experience in the industry, which has been primarily around payments, accounting, controlling and treasury management for both fiat and crypto. I also had some second-hand exposure to trading and investment products.

How it was written

Most of the content was written by my own hand, but I did leverage AI for many things, such as polishing, structuring, research and review.

This book is meant as a living document, and I plan to update it regularly when I find something add-worthy. Feedback is very much welcomed and can be shared anywhere, but GitHub Issues is probably the best place for it.

What to expect

Most of the book should be prefixed with "it depends", as should basically any non-trivial topic. But such a disclaimer serves no purpose. Instead, I try to give safe default advice and enough context for the reader to decide whether a given piece of advice applies to their situation.

The content of the book covers the topics I know something about, which leaves large gaps, e.g. around hardcore trading, pricing, capital markets, and more. Yet the title is generic and aspirational - I would love to cover as much of the fintech space as possible. If you think something important and widely useful is missing, please let me know.

A lot of the material (idempotency, ordering, retries, reconciliation) is general distributed-systems practice, but it's included here because these topics are critical when stakes are high, and a must-know for any fintech engineer.

The book was written with a thinking reader in mind, and I assume no one will follow it blindly. It should go without saying that your internal guidelines and practices take precedence over the advice I share (though the latter could prompt you to question the former). And nothing here is legal, compliance or financial advice. Of course.

Need more help?

If you need direct help or advice with something specific to your fintech system, I do a bit of consulting and will be glad to help!