TypeScript Handbook Definitivo para Blockchain

Do zero (de verdade) até SDKs, APIs, eventos, TX building, e sistemas críticos com segurança

Por que isso importa: O objetivo

Você vai aprender TypeScript como **ferramenta de engenharia** aplicada a blockchain: **modelagem de domínio**, **contratos explícitos**, **validação runtime**, **I/O seguro**, **criptografia/encoding**, **eventos**, **testes** e **arquitetura**.
Fonte oficial (Handbook): https://www.typescriptlang.org/pt/docs/handbook/

Checklist: O que este handbook entrega (promessa)

- Tipos e padrões TS que **reduzem bug em produção** (não só "sintaxe bonita")
- Receitas para SDK/API/eventos com **contratos fortes + validação runtime**
- Guia prático para **crypto hygiene** (BigInt, bytes, hex, base58/bech32, hashes, assinaturas)
- Modelos reais: swaps, wallets, withdrawals, jobs, idempotência, retries, dedup
- Estruturas de projeto: tsconfig, lint, testes, CI, monorepo, release

Sumário

1 Mentalidade correta: TypeScript em blockchain

Definição: A frase que evita 80% dos erros

O runtime é JavaScript. O TypeScript **não valida dados externos sozinho**. Tudo que vem de rede, fila, banco, usuário, node RPC, explorer: **validar em runtime**.

Por que isso importa: Por que blockchain amplifica bugs

Em blockchain, o bug não é "só erro": pode virar **perda de fundos**, **estado irrecuperável**, **inconsistência contábil**, **assinatura inválida**, ou **TX mal formada**. Tipos evitam classes enormes de erro **antes de rodar**. Validação runtime impede lixo entrar.

Erro comum: Anti-padrão fatal

"Se compila, está seguro." Não. Você precisa de:

- **Tipos** (design-time)
- **Validação runtime** (input/IO)
- **Invariantes de domínio** (regras)
- **Observabilidade** (logs/metrics/traces)
- **Testes** (unit + integration + property)

2 Parte I — Fundamentos TS (o suficiente para ser perigoso)

2.1 Inferência e anotações (The Basics)

```ts
let amount = 10; // number
let address = "bc1..."; // string

function sum(a: number, b: number): number {
  return a + b;
}
```

Dica prática: Regra

Deixe TS inferir quando óbvio. Anote quando:

- o contrato é público (SDK/API)
- o retorno é importante (ex.: Result)
- a função é genérica (ex.: parse/validate)

2.2 any, unknown, never

Definição: unknown (padrão correto para input)

`unknown` significa "não confio". Você é obrigado a checar antes de usar.

```ts
function parseNonEmptyString(x: unknown, field = "value"): string {
  if (typeof x === "string" && x.trim().length > 0) return x;
  throw new Error(`${field} invalido`);
}
```

Definição: never (exhaustiveness)

`never` te garante que todos os casos foram tratados.

```
type Network = "btc" | "ln" | "liquid";

function assertNever(x: never): never {
  throw new Error(`Caso nao tratado: ${String(x)}`);
}

function networkPrefix(n: Network): string {
  switch (n) {
    case "btc": return "BTC";
    case "ln": return "LN";
    case "liquid": return "L-BTC";
    default: return assertNever(n);
  }
}
```

Erro comum: any

`any` desliga o TS e destrói garantias. Em blockchain, use `any` só como ponte temporária.

3 Parte II — Modelagem de domínio (o coração)

3.1 Literal types e unions: estados impossíveis viram impossíveis

```
type SwapType = "single" | "split";
type SwapStatus = "created" | "quoting" | "executing" | "done" | "failed";

type SwapId = string & { readonly __brand: "SwapId" };
type TxId = string & { readonly __brand: "TxId" };

type Swap = Readonly<{
  id: SwapId;
  type: SwapType;
  status: SwapStatus;
  amountSats: bigint;
}>;
```

Por que isso importa: Brand types (IDs que não se misturam)

Em sistemas de swaps/wallets, misturar id errado vira bug silencioso. Brand types criam "strings incompatíveis" sem custo em runtime.

```
function asSwapId(x: string): SwapId { return x as SwapId; }
function asTxId(x: string): TxId { return x as TxId; }
```

3.2 Discriminated unions: eventos e jobs à prova de payload errado

```ts
type Job =
  | { kind: "swap.execute"; txId: TxId; split: false }
  | { kind: "swap.execute"; txId: TxId; split: true; splitId: string }
  | { kind: "wallet.withdraw"; walletId: string; amountSats: bigint };

function consume(job: Job) {
  switch (job.kind) {
    case "swap.execute":
      if (job.split) return job.splitId; // garantido
      return null;
    case "wallet.withdraw":
      return job.amountSats;
    default:
      return assertNever(job);
  }
}
```

Dica prática: Regra de ouro para filas

Todo payload de fila deve ter:

- `kind` (discriminador)
- `idempotencyKey` (ou `eventId`)
- `createdAt` (ISO)
- versão (opcional, mas recomendado): `schemaVersion`

4 Parte III — Runtime validation (onde a vida real entra)

Definição: Por que runtime validation é inevitável

TypeScript não valida JSON. Nodes, explorers e APIs retornam qualquer coisa. Logo: **parse + validate + mapear para tipo interno**.

4.1 Padrão: parse/validate → type safe

```ts
type ApiWithdrawal = {
  success: boolean;
  txHash?: string;
  errorMessage?: string;
};

function isRecord(x: unknown): x is Record<string, unknown> {
  return typeof x === "object" && x !== null;
}

function parseApiWithdrawal(x: unknown): ApiWithdrawal {
  if (!isRecord(x)) throw new Error("payload invalido");
  const success = x["success"];
  if (typeof success !== "boolean") throw new Error("success invalido");

  const txHash = x["txHash"];
  if (txHash !== undefined && typeof txHash !== "string") {
    throw new Error("txHash invalido");
  }

  const errorMessage = x["errorMessage"];
  if (errorMessage !== undefined && typeof errorMessage !== "string") {
    throw new Error("errorMessage invalido");
  }

  return { success, txHash, errorMessage };
}
```

Por que isso importa: Por que isso importa

Você "tranca" a fronteira: fora do seu core é `unknown`. Depois do parse, é tipo confiável.

Dica prática: Quando usar bibliotecas (zod/valibot)

Se sua equipe usa muito JSON, schemas e DTOs, usar zod/valibot acelera. Mas entenda o padrão manual: ele é a base de tudo (e evita magia).

5 Parte IV — Result<T,E> (domínio sem exceções como fluxo)

5.1 O padrão profissional: Ok/Err

```
type Ok<T> = { ok: true; value: T };
type Err<E> = { ok: false; error: E };
type Result<T, E> = Ok<T> | Err<E>;

const ok = <T>(value: T): Ok<T> => ({ ok: true, value });
const err = <E>(error: E): Err<E> => ({ ok: false, error });

type DomainError =
  | { kind: "InsufficientFunds"; balanceSats: bigint; neededSats: bigint }
  | { kind: "InvalidAddress"; reason: string }
  | { kind: "NetworkDown"; provider: string }
  | { kind: "RateLimited"; retryAfterMs?: number };

function isOk<T,E>(r: Result<T,E>): r is Ok<T> { return r.ok; }
```

5.2 Compondo Result (map/flatMap)

```
function map<T,E,U>(r: Result<T,E>, f: (x: T) => U): Result<U,E> {
  return r.ok ? ok(f(r.value)) : r;
}

function flatMap<T,E,U>(r: Result<T,E>, f: (x: T) => Result<U,E>): Result<U,E> {
  return r.ok ? f(r.value) : r;
}
```

Por que isso importa: Por que isso importa em blockchain

Você quer diferenciar:

- Erros esperados (domínio): saldo insuficiente, address inválido
- Erros inesperados: bug, null, invariantes quebradas

Result deixa explícito o que é "esperado".

6 Parte V — Bytes, BigInt e encodings (criptografia sem dor)

Definição: O trio que manda em blockchain

(1) bigint, **(2) Uint8Array/Buffer**, **(3) encoding (hex/base58/bech32)**. Quase todo bug sério vem de conversão errada aqui.

6.1 BigInt: satoshis, fees e segurança numérica

Erro comum: Erro comum

Usar `number` para satoshis e somas grandes. `number` tem limite de precisão (IEEE-754).

```
type Sats = bigint & { readonly __brand: "Sats" };
const sats = (x: bigint): Sats => x as Sats;

function addSats(a: Sats, b: Sats): Sats {
```

```
  return sats((a as bigint) + (b as bigint));
}

function feeFromRate(amountSats: Sats, ppm: bigint): Sats {
  // ppm: partes por milhao
  const amt = amountSats as bigint;
  return sats((amt * ppm) / 1_000_000n);
}
```

6.2 Bytes: Buffer vs Uint8Array (Node.js)

```
function hexToBytes(hex: string): Uint8Array {
  if (hex.length % 2 !== 0) throw new Error("hex invalido");
  const out = new Uint8Array(hex.length / 2);
  for (let i = 0; i < out.length; i++) {
    out[i] = parseInt(hex.slice(i*2, i*2+2), 16);
  }
  return out;
}

function bytesToHex(b: Uint8Array): string {
  return Array.from(b).map(x => x.toString(16).padStart(2,"0")).join("");
}
```

Dica prática: Dica prática

No Node, `Buffer` é um `Uint8Array` com extras. Em libs modernas, prefira `Uint8Array`.

6.3 Hash (sha256) de forma consistente

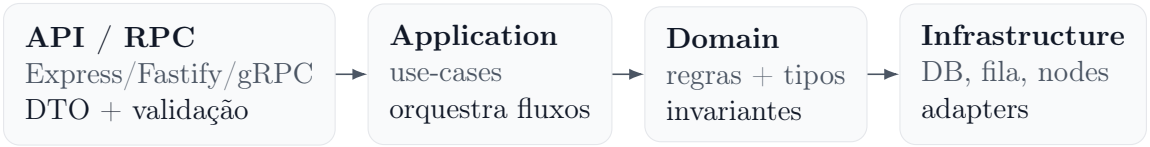
```
import { createHash } from "crypto";

export function sha256Bytes(data: Uint8Array): Uint8Array {
  const h = createHash("sha256").update(data).digest();
  return new Uint8Array(h);
}

export function sha256HexUtf8(s: string): string {
  return createHash("sha256").update(Buffer.from(s, "utf8")).digest("hex");
}
```

7 Parte VI — Arquitetura TS para blockchain: camadas e fronteiras

7.1 O diagrama mínimo (o que funciona em produção)



Por que isso importa: Por que isso importa

Em blockchain, você integra com: node RPC, LN node, explorer, DB, filas, custodians. Separar camadas evita "acoplamento fatal" e permite testes reais.

7.2 Ports & Adapters: interfaces que blindam dependências

```ts
type TxHex = string & { readonly __brand: "TxHex" };
type TxHash = string & { readonly __brand: "TxHash" };

interface BitcoinNodePort {
  broadcast(tx: TxHex): Promise<Result<TxHash, DomainError>>;
  getFeeRate(): Promise<Result<bigint, DomainError>>; // sat/vB ou similar
}

interface WalletRepo {
  getBalance(walletId: string): Promise<Sats>;
  saveWithdrawal(wid: string, txHash: TxHash): Promise<void>;
}
```

Dica prática: Regra

Seu **domínio** não importa libs de HTTP, DB, SDK de node. Domínio fala com interfaces. Infra implementa.

8 Parte VII — Idempotência, retries e filas (onde tudo quebra)

8.1 Idempotência: a lei de ouro de pagamentos

Definição: Idempotência em uma linha

A mesma requisição repetida não pode causar duas saídas de fundos.

```ts
type IdempotencyKey = string & { readonly __brand: "IdempotencyKey" };

type WithdrawalRequest = {
  walletId: string;
  amountSats: Sats;
  toAddress: string;
  idempotencyKey: IdempotencyKey;
};

type WithdrawalState =
  | { status: "received"; req: WithdrawalRequest }
  | { status: "broadcasting"; req: WithdrawalRequest; txHex: TxHex }
  | { status: "broadcasted"; req: WithdrawalRequest; txHash: TxHash }
```

```
  | { status: "failed"; req: WithdrawalRequest; error: DomainError };
```

8.2 Retry com backoff (sem duplicar efeitos)

```
function sleep(ms: number) { return new Promise(r => setTimeout(r, ms)); }

async function retry<T>(fn: () => Promise<Result<T,DomainError>>, attempts = 5) {
  let wait = 200;
  for (let i = 0; i < attempts; i++) {
    const r = await fn();
    if (r.ok) return r;

    // erros "no-retry"
    if (r.error.kind === "InvalidAddress" || r.error.kind === "InsufficientFunds") {
      return r;
    }

    await sleep(wait);
    wait = Math.min(wait * 2, 5000);
  }
  return err<DomainError>({ kind: "NetworkDown", provider: "retry-exhausted" });
}
```

Erro comum: Erro comum em filas

Retry sem idempotência vira duplicação de pagamento. Retry deve ser seguro porque o efeito é protegido por chave idempotente.

9 Parte VIII — TX building (conceitual + tipos para não se cortar)

Por que isso importa: Nota importante

Este handbook foca em **engenharia TypeScript** e padrões. Construção real de TX (PSBT, witness, scripts) depende da lib escolhida. O objetivo aqui é te dar **tipos, fronteiras, invariantes** e um blueprint de implementação.

9.1 Modelo mínimo: UTXO, Output e seleção

```
type OutPoint = Readonly<{ txid: string; vout: number }>;
type ScriptPubKeyHex = string & { readonly __brand: "ScriptPubKeyHex" };

type Utxo = Readonly<{
  outpoint: OutPoint;
  valueSats: Sats;
  scriptPubKey: ScriptPubKeyHex;
  // extras: derivation path, key fingerprint, address, etc.
}>;

type TxOutput = Readonly<{
  toAddress: string;
  valueSats: Sats;
}>;

type CoinSelectResult = Readonly<{
  inputs: readonly Utxo[];
```

```
  outputs: readonly TxOutput[];
  feeSats: Sats;
  change?: TxOutput;
}>;
```

9.2 Seleção de moedas (esqueleto) com invariantes

```ts
function sumUtxos(xs: readonly Utxo[]): Sats {
  let acc = 0n;
  for (const u of xs) acc += (u.valueSats as bigint);
  return sats(acc);
}

function sumOutputs(xs: readonly TxOutput[]): Sats {
  let acc = 0n;
  for (const o of xs) acc += (o.valueSats as bigint);
  return sats(acc);
}

type CoinSelectError =
  | { kind: "NotEnoughFunds"; have: Sats; need: Sats }
  | { kind: "InvalidFee"; reason: string };

function coinSelectSimple(
  utxos: readonly Utxo[],
  outputs: readonly TxOutput[],
  feeSats: Sats
): Result<CoinSelectResult, CoinSelectError> {
  const have = sumUtxos(utxos);
  const need = sats((sumOutputs(outputs) as bigint) + (feeSats as bigint));
  if ((have as bigint) < (need as bigint)) {
    return err({ kind: "NotEnoughFunds", have, need });
  }
  // simplificado: usa todos os utxos e change fixo
  const changeValue = sats((have as bigint) - (need as bigint));
  const change = (changeValue as bigint) > 0n
    ? { toAddress: "CHANGE_ADDRESS", valueSats: changeValue } as TxOutput
    : undefined;

  return ok({ inputs: utxos, outputs, feeSats, change });
}
```

Dica prática: O que você melhoraria em produção

- dust threshold
- seleção eficiente (min inputs, privacy, avoid merging)
- fee rate (sat/vB) e estimativa de vsize
- change address derivado (HD)
- políticas (consolidation windows)

10 Parte IX — SDK/API: contratos públicos que não quebram

10.1 DTO externo vs modelo interno

Definição: Regra

DTO é o que entra/sai da rede. Modelo interno é o que seu core usa. Nunca misture.

```
type WithdrawDTO = {
  walletId: string;
  amountSats: string; // em APIs, string para BigInt
  toAddress: string;
  idempotencyKey: string;
};

type WithdrawCommand = {
  walletId: string;
  amountSats: Sats;
  toAddress: string;
  idempotencyKey: IdempotencyKey;
};

function parseWithdrawCommand(dto: unknown): WithdrawCommand {
  if (!isRecord(dto)) throw new Error("dto invalido");
  const walletId = parseNonEmptyString(dto.walletId, "walletId");
  const toAddress = parseNonEmptyString(dto.toAddress, "toAddress");
  const idk = parseNonEmptyString(dto.idempotencyKey, "idempotencyKey") as IdempotencyKey;

  const amountStr = parseNonEmptyString(dto.amountSats, "amountSats");
  if (!/^\d+$/.test(amountStr)) throw new Error("amountSats deve ser inteiro");
  const amountSats = sats(BigInt(amountStr));

  return { walletId, toAddress, idempotencyKey: idk, amountSats };
}
```

Erro comum: Erro comum

API usando `number` para satoshis e timestamps. Em JSON, use `string` e converta.

11 Parte X — tsconfig (modo engenharia)

11.1 Config recomendado (base para produção)

```
{
  "compilerOptions": {
    "target": "ES2022",
    "moduleResolution": "node16",
    "module": "Node16",

    "strict": true,
    "noUncheckedIndexedAccess": true,
    "exactOptionalPropertyTypes": true,
    "useUnknownInCatchVariables": true,

    "forceConsistentCasingInFileNames": true,
    "noImplicitOverride": true,
    "noFallthroughCasesInSwitch": true,

    "skipLibCheck": true
  }
}
```

Por que isso importa: Por que essas flags importam

`noUncheckedIndexedAccess` e `exactOptionalPropertyTypes` eliminam classes inteiras de bug com JSON/DTO.

12 Parte XI — Testes (unit, integration e "invariantes")

12.1 Testes de domínio (rápidos e baratos)

```
function assert(cond: unknown, msg = "assert failed"): asserts cond {
  if (!cond) throw new Error(msg);
}

function test_coinselect_not_enough() {
  const utxos: Utxo[] = [
    { outpoint: { txid: "a", vout: 0 }, valueSats: sats(1000n), scriptPubKey: "00" as
        ScriptPubKeyHex }
  ];
  const outputs: TxOutput[] = [{ toAddress: "bc1xxx", valueSats: sats(2000n) }];
  const r = coinSelectSimple(utxos, outputs, sats(10n));
  assert(!r.ok, "deveria falhar");
  assert(r.error.kind === "NotEnoughFunds");
}
```

12.2 Testes de integração (com nodes/containers)

Dica prática: Dica prática

Integração de blockchain não é "mockar tudo". Tenha um profile de testes que sobe serviços (docker-compose) e valida fluxo real: broadcast fake, getBalance, getTx, mempool.

13 Parte XII — Segurança prática (onde TS ajuda muito)

Checklist: Regras de ouro

- Nunca confie em input externo: `unknown` + parse/validate
- Não use `number` para dinheiro: `bigint`
- Separe DTO (JSON) do core (tipado)
- Idempotência para endpoints que movem fundos
- Logs sem vazar segredos (seed, xprv, tokens)
- Erros de domínio explícitos (Result) e métricas para falhas

Erro comum: Erro comum em backend de wallet

Logar payload inteiro (inclui secrets, addresses, memos). Crie um `redact()` central.

```
function redact(x: Record<string, unknown>) {
  const copy: Record<string, unknown> = { ...x };
  for (const k of ["seed", "xprv", "mnemonic", "privateKey", "token"]) {
    if (k in copy) copy[k] = "[REDACTED]";
  }
  return copy;
}
```

14 Parte XIII — Exercícios (para virar "fluente")

Exercício rápido: Exercício 1: Result + DomainError

Crie um `validateAddress(network, address)` que retorna `Result<true, DomainError>`.

Exercício rápido: Exercício 2: Job discriminado + versionamento

Adicione `schemaVersion: 1` em todos os jobs. Garanta via tipo que não existe job sem versão.

Exercício rápido: Exercício 3: DTO parse

Implemente `parseDepositRequest(dto: unknown)` que valida: `walletId`, `network`, `amountSats` (string numérica), `memo?`.

Exercício rápido: Exercício 4: invariantes

Crie uma função `assertSatsPositive(x)` que lança se `x <= 0`. Use em todos os pontos críticos (withdraw, send, fee).

15 Parte XIV — Ethers v6 (EVM) em modo engenharia (SDK + backend)

Por que isso importa: Objetivo

Construir um módulo EVM profissional em TypeScript com:

- Providers e fallback (resiliência)
- ABI typing (contratos fortemente tipados)
- Events tipados (decode seguro)
- Retries, rate-limit e timeouts
- Nonce management (concorrência)
- Multicall (batch eficiente)
- Safe tx submission (simulação, gas, replacement, confirmação)

15.1 Instalação e postura (v6)

```
// npm i ethers@^6
import { ethers } from "ethers";
```

Erro comum: Erro comum

Misturar mentalidade v5/v6 (tipos e APIs mudaram). Trabalhe sempre com **BigInt** no v6.

15.2 Provider resiliente (fallback + timeouts + retry)

Definição: Padrão

Não confie em 1 RPC. Use **fallback** e **retry** para erros transitórios.

```
type RpcUrl = string & { readonly __brand: "RpcUrl" };

type RpcConfig = Readonly<{
  chainId: number;
  rpcs: readonly RpcUrl[];
  timeoutMs: number;
  maxAttempts: number;
}>;

type RpcError =
  | { kind: "Timeout"; url: string }
  | { kind: "RateLimited"; url: string; retryAfterMs?: number }
  | { kind: "Network"; url: string; message: string };

function withTimeout<T>(p: Promise<T>, ms: number, url: string): Promise<T> {
  return new Promise((resolve, reject) => {
    const t = setTimeout(() => reject({ kind: "Timeout", url } satisfies RpcError), ms);
    p.then(x => { clearTimeout(t); resolve(x); }, e => { clearTimeout(t); reject(e); });
  });
}

function isRateLimit(e: any): { retryAfterMs?: number } | null {
  const msg = String(e?.message ?? e);
  if (msg.includes("429") || msg.toLowerCase().includes("rate limit")) return {};
  return null;
}

async function rpcCall<T>(
  urls: readonly string[],
  timeoutMs: number,
  attempts: number,
  fn: (provider: ethers.JsonRpcProvider) => Promise<T>
): Promise<T> {
  let lastErr: unknown;
  for (let a = 0; a < attempts; a++) {
    for (const url of urls) {
      try {
        const p = new ethers.JsonRpcProvider(url);
        return await withTimeout(fn(p), timeoutMs, url);
      } catch (e: any) {
        lastErr = e;
        const rl = isRateLimit(e);
        if (rl) {
          const wait = rl.retryAfterMs ?? 400;
          await new Promise(r => setTimeout(r, wait));
          continue;
        }
        // segue para o prximo RPC
      }
    }
    await new Promise(r => setTimeout(r, Math.min(250 * (a + 1), 1500)));
  }
  throw lastErr;
```

```
}

export function makeResilientProvider(cfg: RpcConfig) {
  return {
    chainId: cfg.chainId,
    call: async <T>(fn: (p: ethers.JsonRpcProvider) => Promise<T>) =>
      rpcCall(cfg.rpcs as unknown as string[], cfg.timeoutMs, cfg.maxAttempts, fn)
  };
}
```

Dica prática: Dica prática

Se você precisa de WebSocket para eventos, use WS como **otimização**, mas mantenha HTTP como **fonte de verdade** com re-scan por bloco.

15.3 ABI typing (contratos tipados sem magia excessiva)

Definição: Estratégia realista

TypeScript não "infere ABI" sozinho. O caminho sólido:

- ABI como `const` (para literais)
- Tipos de entrada/saída definidos manualmente para funções críticas
- Um wrapper com interface estável (SDK)

```
const erc20Abi = [
  "function balanceOf(address) view returns (uint256)",
  "function transfer(address to, uint256 value) returns (bool)",
  "event Transfer(address indexed from, address indexed to, uint256 value)"
] as const;

type Address = string & { readonly __brand: "Address" };
type TxHash = string & { readonly __brand: "TxHash" };

function asAddress(x: string): Address { return x as Address; }

export class ERC20Client {
  private readonly contract: ethers.Contract;

  constructor(
    private readonly provider: ethers.Provider,
    private readonly token: Address
  ) {
    this.contract = new ethers.Contract(token, erc20Abi, provider);
  }

  async balanceOf(holder: Address): Promise<bigint> {
    return await this.contract.balanceOf(holder) as bigint;
  }

  connectSigner(signer: ethers.Signer) {
    return new ERC20SignerClient(signer, this.token);
  }
}

export class ERC20SignerClient {
  private readonly contract: ethers.Contract;
```

```
  constructor(
    private readonly signer: ethers.Signer,
    token: Address
  ) {
    this.contract = new ethers.Contract(token, erc20Abi, signer);
  }

  async transfer(to: Address, value: bigint): Promise<TxHash> {
    const tx = await this.contract.transfer(to, value);
    return tx.hash as TxHash;
  }
}
```

15.4 Eventos tipados (decode seguro + reorg safety)

Definição: Reorg safety

Nunca trate evento como final antes de **N confirmações**. Sempre salve: `blockNumber`, `txHash`, `logIndex`, `blockHash`.

```
type TransferEvent = Readonly<{
  from: Address;
  to: Address;
  value: bigint;
  blockNumber: number;
  txHash: TxHash;
  logIndex: number;
  blockHash: string;
}>;

function parseTransferLog(log: ethers.Log): TransferEvent | null {
  const iface = new ethers.Interface(erc20Abi);
  try {
    const parsed = iface.parseLog(log);
    if (parsed.name !== "Transfer") return null;
    const from = parsed.args[0] as string;
    const to = parsed.args[1] as string;
    const value = parsed.args[2] as bigint;
    return {
      from: asAddress(from),
      to: asAddress(to),
      value,
      blockNumber: log.blockNumber ?? 0,
      txHash: log.transactionHash as TxHash,
      logIndex: log.index,
      blockHash: log.blockHash ?? ""
    };
  } catch {
    return null;
  }
}
```

15.5 Nonce management (concorrência real)

Erro comum: Erro comum

Enviar 2 txs em paralelo com o mesmo signer usando `getTransactionCount` sem lock: resultado: **nonce collisions**, replacement aleatório, stuck tx.

```
class Mutex {
  private busy = Promise.resolve();
  run<T>(fn: () => Promise<T>): Promise<T> {
    const next = this.busy.then(fn, fn);
    this.busy = next.then(() => undefined, () => undefined);
    return next;
  }
}

type NonceState = { next: number; updatedAtMs: number };

export class NonceManager {
  private readonly mu = new Mutex();
  private state: NonceState | null = null;

  constructor(private readonly signer: ethers.Signer, private readonly ttlMs = 10_000) {}

  private async refresh(): Promise<NonceState> {
    const addr = await this.signer.getAddress();
    const n = await this.signer.provider!.getTransactionCount(addr, "pending");
    return { next: n, updatedAtMs: Date.now() };
  }

  async allocate(): Promise<number> {
    return this.mu.run(async () => {
      const now = Date.now();
      if (!this.state || (now - this.state.updatedAtMs) > this.ttlMs) {
        this.state = await this.refresh();
      }
      const n = this.state.next;
      this.state.next += 1;
      return n;
    });
  }
}
```

15.6 Multicall (batch eficiente)

Definição: Estratégia

Você pode:

- usar contrato Multicall (per chain)
- ou fazer `eth_call` paralelo com limite de concorrência

Abaixo: versão pragmática com limite de concorrência.

```
async function mapLimit<T,U>(xs: readonly T[], limit: number, fn: (x: T) => Promise<U>):
    Promise<U[]> {
  const out: U[] = [];
  let i = 0;
  const workers = Array.from({ length: Math.min(limit, xs.length) }, async () => {
    while (i < xs.length) {
      const idx = i++;
      out[idx] = await fn(xs[idx]);
    }
  });
  await Promise.all(workers);
  return out;
}

async function batchBalanceOf(
  provider: ethers.Provider,
  token: Address,
  holders: readonly Address[]
): Promise<bigint[]> {
  const c = new ethers.Contract(token, erc20Abi, provider);
  return mapLimit(holders, 8, (h) => c.balanceOf(h) as Promise<bigint>);
}
```

15.7 Safe tx submission (simulação, gas, replacement, confirmações)

Checklist: Antes de enviar

- Simular: `callStatic` quando possível
- Estimar gas e aplicar margem
- Definir maxFee/maxPriority (EIP-1559)
- Persistir **idempotencyKey** e **nonce** no DB
- Confirmar com N blocos (reorg safe)

```ts
type SubmitTxResult = Readonly<{
  txHash: TxHash;
  nonce: number;
}>;

async function safeTransferErc20(
  signer: ethers.Signer,
  nm: NonceManager,
  token: Address,
  to: Address,
  value: bigint
): Promise<SubmitTxResult> {
  const c = new ethers.Contract(token, erc20Abi, signer);

  // (1) simulao (se falhar, no envia)
  await c.transfer.staticCall(to, value);

  // (2) nonce alocado sob lock
  const nonce = await nm.allocate();

  // (3) estimate + margem
  const est = await c.transfer.estimateGas(to, value, { nonce });
  const gasLimit = est + (est / 5n);

  // (4) fees (pragmtico)
  const fee = await signer.provider!.getFeeData();
  const maxFeePerGas = fee.maxFeePerGas ?? 0n;
  const maxPriorityFeePerGas = fee.maxPriorityFeePerGas ?? 0n;

  const tx = await c.transfer(to, value, {
    nonce,
    gasLimit,
    maxFeePerGas,
    maxPriorityFeePerGas
  });

  return { txHash: tx.hash as TxHash, nonce };
}
```

16 Parte XV — Bitcoin (PSBT) em TS: scripts, UTXO selection, RBF/CPFP

Por que isso importa: Objetivo

Criar um módulo Bitcoin profissional (conceitos + implementação TS) com:

- Tipos para scripts/endereços/utxos
- Seleção avançada (coin control, privacidade, custo)
- Dust e regras de saída
- Fee rate e estimativa de vsize
- RBF (replacement) e CPFP (child pays for parent)

Dica prática: Bibliotecas

Para PSBT real, use uma lib de Bitcoin (ex.: `bitcoinjs-lib` ou stack equivalente). Aqui focamos em **modelagem e invariantes**, e deixamos os detalhes de assinatura para o adapter da lib.

16.1 Tipos fundamentais

```ts
type Hex = string & { readonly __brand: "Hex" };
type TxId = string & { readonly __brand: "BtcTxId" };
type Address = string & { readonly __brand: "BtcAddress" };

type Vout = number & { readonly __brand: "Vout" };
type Sats = bigint & { readonly __brand: "Sats" };
const sats = (x: bigint): Sats => x as Sats;

type OutPoint = Readonly<{ txid: TxId; vout: Vout }>;

type ScriptPubKey = Hex & { readonly __brand: "ScriptPubKey" };

type Utxo = Readonly<{
  outpoint: OutPoint;
  valueSats: Sats;
  scriptPubKey: ScriptPubKey;
  // hints de coin control / privacidade:
  account?: string;
  isChange?: boolean;
  confirmations?: number;
  label?: string;
}>;

type TxOutput = Readonly<{ address: Address; valueSats: Sats }>;
```

16.2 Dust e políticas

Definição: Dust

Saída pequena demais vira poeira: difícil de gastar e desperdiça fee. Em produção, você aplica um **dust threshold** (depende do script).

```
type DustPolicy = Readonly<{
  // simplificado: threshold fixo (em produo, por script-type)
  minOutputSats: Sats;
}>;

function isDust(value: Sats, policy: DustPolicy): boolean {
  return (value as bigint) < (policy.minOutputSats as bigint);
}
```

16.3 Fee rate e vsize (estimativa pragmática)

Definição: Vsize

O custo real depende do tipo de input (p2wpkh, p2tr, etc.). Você pode usar:

- estimativas por tipo (aprox)
- ou PSBT builder da lib e medir vsize real

Aqui: estimativa por tipos com envelopes.

```
type ScriptType = "p2wpkh" | "p2tr" | "p2sh-p2wpkh" | "unknown";

type UtxoTyped = Utxo & Readonly<{ scriptType: ScriptType }>;

type FeeRate = bigint & { readonly __brand: "SatPerVb" }; // sat/vB
const feeRate = (x: bigint): FeeRate => x as FeeRate;

function estInputVbytes(t: ScriptType): bigint {
  switch (t) {
    case "p2wpkh": return 68n; // approx
    case "p2tr": return 58n; // approx key-path
    case "p2sh-p2wpkh": return 91n; // approx
    default: return 110n;
  }
}

function estOutputVbytes(): bigint { return 31n; } // p2wpkh-ish approx
function estOverheadVbytes(): bigint { return 10n; } // version/locktime/etc.

function estimateFee(
  inputs: readonly UtxoTyped[],
  outputsCount: number,
  fr: FeeRate
): Sats {
  const vin = inputs.reduce((a, u) => a + estInputVbytes(u.scriptType), 0n);
  const vout = BigInt(outputsCount) * estOutputVbytes();
  const vbytes = estOverheadVbytes() + vin + vout;
  return sats(vbytes * (fr as bigint));
}
```

16.4 Seleção avançada: coin control, privacidade, custo

Definição: Estratégia de seleção (pragmática)

Prioridade típica:

- UTXOs confirmados (se possível)
- evitar misturar "clusters" (privacidade)
- minimizar inputs (fee)
- evitar criar dust no troco

```ts
type SelectionPolicy = Readonly<{
  feeRate: FeeRate;
  dust: DustPolicy;
  minConfirmations: number;
  avoidMixingAccounts?: boolean;
  preferFewerInputs?: boolean;
  maxInputs?: number;
}>;

type CoinSelectError =
  | { kind: "NotEnoughFunds"; have: Sats; need: Sats }
  | { kind: "TooManyInputs"; inputs: number; max: number }
  | { kind: "DustChange"; change: Sats };

type CoinSelectResult = Readonly<{
  inputs: readonly UtxoTyped[];
  outputs: readonly TxOutput[];
  feeSats: Sats;
  change?: TxOutput;
  vbytesEst: bigint;
}>;

function sumUtxos(xs: readonly Utxo[]): Sats {
  let acc = 0n;
  for (const u of xs) acc += (u.valueSats as bigint);
  return sats(acc);
}
function sumOutputs(xs: readonly TxOutput[]): Sats {
  let acc = 0n;
  for (const o of xs) acc += (o.valueSats as bigint);
  return sats(acc);
}

function coinSelect(
  utxos: readonly UtxoTyped[],
  outputs: readonly TxOutput[],
  changeAddr: Address,
  policy: SelectionPolicy
): Result<CoinSelectResult, CoinSelectError> {
  const target = sumOutputs(outputs);

  // filtro por confirmaes
  let candidates = utxos.filter(u => (u.confirmations ?? 0) >= policy.minConfirmations);

  // opcional: evitar mixing por "account"
  if (policy.avoidMixingAccounts) {
    const acc = candidates.find(u => u.account)?.account;
    if (acc) candidates = candidates.filter(u => u.account === acc);
```

```
  }

  // ordenao
  candidates = [...candidates].sort((a,b) => {
    // preferir fewer inputs: maiores primeiro
    if (policy.preferFewerInputs) return Number((b.valueSats as bigint) - (a.valueSats as
        bigint));
    // seno: menores primeiro (gasta poeira, consolida)
    return Number((a.valueSats as bigint) - (b.valueSats as bigint));
  });

  const picked: UtxoTyped[] = [];
  let have = 0n;

  for (const u of candidates) {
    picked.push(u);
    have += (u.valueSats as bigint);

    const feeNoChange = estimateFee(picked, outputs.length, policy.feeRate);
    const needNoChange = (target as bigint) + (feeNoChange as bigint);

    if (have >= needNoChange) {
      // tenta incluir change
      const feeWithChange = estimateFee(picked, outputs.length + 1, policy.feeRate);
      const needWithChange = (target as bigint) + (feeWithChange as bigint);
      const change = have - needWithChange;

      const vbytesEst = (feeWithChange as bigint) / (policy.feeRate as bigint);

      if (change <= 0n) {
        // sem change: ok
        const vbytesNoChange = (feeNoChange as bigint) / (policy.feeRate as bigint);
        return ok({
          inputs: picked,
          outputs,
          feeSats: feeNoChange,
          change: undefined,
          vbytesEst: vbytesNoChange
        });
      }

      const changeSats = sats(change);
      if (isDust(changeSats, policy.dust)) {
        return err({ kind: "DustChange", change: changeSats });
      }

      const changeOut: TxOutput = { address: changeAddr, valueSats: changeSats };
      const max = policy.maxInputs ?? 200;
      if (picked.length > max) return err({ kind: "TooManyInputs", inputs: picked.length, max
          });

      return ok({
        inputs: picked,
        outputs,
        feeSats: feeWithChange,
        change: changeOut,
        vbytesEst
      });
    }
  }
```

```
  const haveSats = sats(have);
  // estimativa grosseira do que faltou (assume 1 input a mais)
  const feeGuess = estimateFee(picked, outputs.length + 1, policy.feeRate);
  const needGuess = sats((target as bigint) + (feeGuess as bigint));
  return err({ kind: "NotEnoughFunds", have: haveSats, need: needGuess });
}
```

16.5 RBF e CPFP (modelo de decisão)

Definição: RBF

Trocar a tx por outra com fee maior usando o mesmo nonce (Bitcoin: sequence opt-in).

Definição: CPFP

Criar uma child tx gastando um output da parent com fee alto para "puxar" confirmação.

```
type MempoolTx = Readonly<{
  txid: TxId;
  feeSats: Sats;
  vbytes: bigint;
  rbfOptIn: boolean;
  confirmations: number;
}>;

type BumpStrategy =
  | { kind: "RBF"; newFeeRate: FeeRate }
  | { kind: "CPFP"; childFeeRate: FeeRate; spendOutputVout: number };

function chooseBumpStrategy(tx: MempoolTx): BumpStrategy | null {
  if (tx.confirmations > 0) return null;
  if (tx.rbfOptIn) {
    return { kind: "RBF", newFeeRate: feeRate(30n) };
  }
  // sem RBF: tenta CPFP
  return { kind: "CPFP", childFeeRate: feeRate(50n), spendOutputVout: 0 };
}
```

17 Parte XVI — Lightning: invoices, HTLC lifecycle, idempotência e reconciliação

Por que isso importa: Objetivo

Modelar Lightning como **máquina de estados**:

- invoices e pagamentos
- HTLC lifecycle em tipos
- idempotência
- reconciliação com o nó (source of truth)

17.1 Tipos essenciais (Invoice/Payment)

```ts
type PaymentHash = string & { readonly __brand: "PaymentHash" };
type PaymentPreimage = string & { readonly __brand: "PaymentPreimage" };
type Bolt11 = string & { readonly __brand: "Bolt11" };
type Msat = bigint & { readonly __brand: "Msat" };
const msat = (x: bigint): Msat => x as Msat;

type LnInvoice = Readonly<{
  bolt11: Bolt11;
  paymentHash: PaymentHash;
  amountMsat?: Msat; // invoice pode ser "amountless"
  expiresAt: string; // ISO
  memo?: string;
}>;

type LnPaymentAttempt = Readonly<{
  attemptId: string;
  createdAt: string;
  routeHint?: string;
}>;

type LnPaymentState =
  | { status: "created"; invoice: LnInvoice; idempotencyKey: string }
  | { status: "inflight"; invoice: LnInvoice; attempt: LnPaymentAttempt }
  | { status: "succeeded"; invoice: LnInvoice; preimage: PaymentPreimage; settledAt: string }
  | { status: "failed"; invoice: LnInvoice; reason: string; failedAt: string }
  | { status: "expired"; invoice: LnInvoice; expiredAt: string };
```

17.2 HTLC lifecycle como evento

Definição: Por que eventos

Lightning tem eventos assíncronos e reorder: **subscribe** + **poll** + **reconcile**. Modelar como eventos torna isso robusto.

```
type HtlcEvent =
  | { kind: "htlc.offered"; paymentHash: PaymentHash; chanId: string; amountMsat: Msat; at:
      string }
  | { kind: "htlc.settled"; paymentHash: PaymentHash; preimage: PaymentPreimage; at: string }
  | { kind: "htlc.failed"; paymentHash: PaymentHash; reason: string; at: string };

function applyHtlcEvent(state: LnPaymentState, ev: HtlcEvent): LnPaymentState {
  switch (state.status) {
    case "created":
    case "inflight":
      if (ev.kind === "htlc.settled" && ev.paymentHash === state.invoice.paymentHash) {
        return { status: "succeeded", invoice: state.invoice, preimage: ev.preimage, settledAt:
            ev.at };
      }
      if (ev.kind === "htlc.failed" && ev.paymentHash === state.invoice.paymentHash) {
        return { status: "failed", invoice: state.invoice, reason: ev.reason, failedAt: ev.at
          };
      }
      return state;
    default:
      return state;
  }
}
```

17.3 Idempotência e reconciliação (o que salva produção)

Checklist: Regras

- Cada pagamento tem `idempotencyKey`
- Você persiste estado local + tentativas
- Periodicamente você reconcilia com o nó LN (source of truth)
- Se houver divergência, você corrige o estado local (event sourcing ajuda)

```
interface LightningNodePort {
  payInvoice(bolt11: Bolt11): Promise<{ paymentHash: PaymentHash; attemptId: string }>;
  lookupPayment(hash: PaymentHash): Promise<
    | { status: "succeeded"; preimage: PaymentPreimage; settledAt: string }
    | { status: "failed"; reason: string; failedAt: string }
    | { status: "pending" }
    | { status: "unknown" }
  >;
}

async function reconcilePayment(
  node: LightningNodePort,
  state: LnPaymentState
): Promise<LnPaymentState> {
  const h = state.invoice.paymentHash;
  const remote = await node.lookupPayment(h);
  if (remote.status === "succeeded" && state.status !== "succeeded") {
    return { status: "succeeded", invoice: state.invoice, preimage: remote.preimage, settledAt:
        remote.settledAt };
  }
  if (remote.status === "failed" && state.status !== "failed") {
    return { status: "failed", invoice: state.invoice, reason: remote.reason, failedAt: remote.
       failedAt };
  }
  return state;
}
```

18 Parte XVII — Liquid: assets, peg-in/out (conceitual), custódia e auditoria

Por que isso importa: Objetivo

Liquid exige modelagem de:

- assets (LBTC + outros)
- políticas de custódia (quórum, limites, aprovação)
- trilha de auditoria (who/what/when)
- peg-in/out (conceitual, pois depende do provedor)

18.1 Tipos para assets e saldos

```
type AssetId = string & { readonly __brand: "AssetId" };
type LBTC = Sats & { readonly __brand: "LBTC" };

type AssetAmount = Readonly<{
  asset: AssetId;
  amountSats: Sats; // ou bigint genrico; aqui simplificamos
}>;

type LiquidAddress = string & { readonly __brand: "LiquidAddress" };

type LiquidTransfer = Readonly<{
  id: string;
  fromVault: string;
  toAddress: LiquidAddress;
  asset: AssetId;
  amount: Sats;
  memo?: string;
}>;
```

18.2 Política de custódia e aprovações (modelo implementável)

Definição: Por que isso importa

Em infra institucional, a regra não é só "assinar e mandar". Você precisa: limites, 4-olhos, segregação de função, auditoria.

```
type ActorId = string & { readonly __brand: "ActorId" };

type Approval = Readonly<{
  actor: ActorId;
  at: string;
  decision: "approve" | "reject";
  reason?: string;
}>;

type CustodyPolicy = Readonly<{
  vaultId: string;
  asset: AssetId;
  dailyLimitSats: Sats;
  requireApprovals: number; // M-of-N lgico
  approverSet: readonly ActorId[];
}>;

type TransferState =
  | { status: "requested"; tx: LiquidTransfer; approvals: readonly Approval[]; requestedAt:
      string }
  | { status: "approved"; tx: LiquidTransfer; approvals: readonly Approval[]; approvedAt:
      string }
  | { status: "broadcasted"; tx: LiquidTransfer; txid: string; broadcastedAt: string }
  | { status: "settled"; tx: LiquidTransfer; txid: string; settledAt: string }
  | { status: "rejected"; tx: LiquidTransfer; approvals: readonly Approval[]; rejectedAt:
      string; reason: string };

function canApprove(actor: ActorId, policy: CustodyPolicy): boolean {
  return policy.approverSet.includes(actor);
}

function applyApproval(state: TransferState, policy: CustodyPolicy, a: Approval):
    TransferState {
  if (state.status !== "requested") return state;
  if (!canApprove(a.actor, policy)) return state;

  const approvals = [...state.approvals, a].filter((v, i, arr) =>
    i === arr.findIndex(x => x.actor === v.actor) // 1 voto por ator
  );

  const rejects = approvals.filter(x => x.decision === "reject");
  if (rejects.length > 0) {
    return { status: "rejected", tx: state.tx, approvals, rejectedAt: a.at, reason: rejects[0].
        reason ?? "rejected" };
  }

  const approves = approvals.filter(x => x.decision === "approve");
  if (approves.length >= policy.requireApprovals) {
    return { status: "approved", tx: state.tx, approvals, approvedAt: a.at };
  }

  return { ...state, approvals };
```

```
}
```

18.3 Peg-in/out (conceitual)

Dica prática: Conceito

Peg-in: BTC → LBTC (bridge federada). Peg-out: LBTC → BTC. Em produção, isso passa por provedor/infra. O que você modela é:

- pedido (request)
- evidências (proofs, txids)
- estados (pending, confirmed, failed)
- auditoria

```
type PegState =
  | { status: "requested"; direction: "peg-in" | "peg-out"; amount: Sats; requestedAt: string
      }
  | { status: "pending"; direction: "peg-in" | "peg-out"; amount: Sats; refTxId?: string;
      pendingAt: string }
  | { status: "completed"; direction: "peg-in" | "peg-out"; amount: Sats; refTxId: string;
      completedAt: string }
  | { status: "failed"; direction: "peg-in" | "peg-out"; amount: Sats; reason: string;
      failedAt: string };
```

19 Parte XVIII — Arquitetura completa: event sourcing, outbox, exactly-once pragmático

Por que isso importa: Objetivo

Montar um núcleo transacional para sistemas de swaps/wallets com:

- Event sourcing (audit + replay)
- Outbox pattern (DB → fila sem perder)
- Exactly-once pragmático (idempotência + dedup)
- Reconciliação contábil (ledger)
- Invariantes como código

19.1 Event sourcing: eventos como fonte de verdade

Definição: Modelo

Estado = fold(eventos). Banco guarda: **(a)** eventos imutáveis, **(b)** snapshot/materialized view.

```
type EventId = string & { readonly __brand: "EventId" };
type AggregateId = string & { readonly __brand: "AggregateId" };

type DomainEvent =
  | { kind: "withdrawal.requested"; id: EventId; agg: AggregateId; at: string; amountSats:
      Sats; to: string; idem: string }
  | { kind: "withdrawal.broadcasted"; id: EventId; agg: AggregateId; at: string; txHash:
      string }
  | { kind: "withdrawal.settled"; id: EventId; agg: AggregateId; at: string; confirmations:
      number }
  | { kind: "withdrawal.failed"; id: EventId; agg: AggregateId; at: string; reason: string };

type WithdrawalState =
  | { status: "none" }
  | { status: "requested"; amountSats: Sats; to: string; idem: string }
  | { status: "broadcasted"; amountSats: Sats; to: string; idem: string; txHash: string }
  | { status: "settled"; amountSats: Sats; to: string; txHash: string; confirmations: number }
  | { status: "failed"; amountSats: Sats; to: string; reason: string };

function foldWithdrawal(state: WithdrawalState, ev: DomainEvent): WithdrawalState {
  switch (ev.kind) {
    case "withdrawal.requested":
      return { status: "requested", amountSats: ev.amountSats, to: ev.to, idem: ev.idem };
    case "withdrawal.broadcasted":
      if (state.status !== "requested") return state;
      return { status: "broadcasted", amountSats: state.amountSats, to: state.to, idem: state.
          idem, txHash: ev.txHash };
    case "withdrawal.settled":
      if (state.status !== "broadcasted") return state;
      return { status: "settled", amountSats: state.amountSats, to: state.to, txHash: state.
          txHash, confirmations: ev.confirmations };
    case "withdrawal.failed":
      if (state.status === "none") return state;
      return { status: "failed", amountSats: state.amountSats, to: state.to, reason: ev.reason
          };
    default:
      return state;
  }
```

```
}
```

19.2 Outbox pattern (DB → fila com garantia)

Definição: Por que

Você não quer: salvar no DB e falhar ao publicar evento (ou vice-versa). Outbox resolve publicando a partir de uma tabela transacional.

```
type OutboxRow = Readonly<{
  id: string;
  topic: string;
  payloadJson: string;
  createdAt: string;
  publishedAt?: string;
}>;

interface DbTx {
  insertEvent(ev: DomainEvent): Promise<void>;
  insertOutbox(row: OutboxRow): Promise<void>;
  commit(): Promise<void>;
  rollback(): Promise<void>;
}

async function persistWithOutbox(tx: DbTx, ev: DomainEvent, topic: string) {
  try {
    await tx.insertEvent(ev);
    await tx.insertOutbox({
      id: String(ev.id),
      topic,
      payloadJson: JSON.stringify(ev),
      createdAt: ev.at
    });
    await tx.commit();
  } catch (e) {
    await tx.rollback();
    throw e;
  }
}
```

19.3 Exactly-once pragmático: idempotência + dedup

Definição: A verdade

Exactly-once real é caro/complexo. O pragmático que funciona: **at-least-once** + **dedup** + **idempotência** + **reconcile**.

```
interface InboxRepo {
  hasProcessed(messageId: string): Promise<boolean>;
  markProcessed(messageId: string): Promise<void>;
}

async function consumeOnce(
  inbox: InboxRepo,
  messageId: string,
  handler: () => Promise<void>
) {
  if (await inbox.hasProcessed(messageId)) return;
  await handler();
  await inbox.markProcessed(messageId);
}
```

19.4 Reconciliação contábil (ledger como invariantes)

Definição: Ledger

Tudo que mexe com fundos vira lançamentos de débito/crédito com: `txId`, `asset`, `amount`, `accounts`, `timestamp`.

```
type LedgerAccount =
  | "customer.available"
  | "customer.pending"
  | "vault.hot"
  | "vault.cold"
  | "fee.revenue"
  | "network.fees";

type LedgerEntry = Readonly<{
  entryId: string;
  at: string;
  asset: string; // BTC/LBTC/USDT...
  amount: bigint; // sempre positivo no entry
  debit: LedgerAccount;
  credit: LedgerAccount;
  ref: string; // txId / swapId / paymentHash
}>;

function validateBalanced(entries: readonly LedgerEntry[]): boolean {
  // Em double-entry, cada entry individual j  balanceada.
  // Validaes reais: somatrios por ref, limites, etc.
  return entries.every(e => e.amount > 0n);
}
```

20 Parte XIX — Monorepo + releases: core, sdk, api, infra, versionamento e CI/CD

Por que isso importa: Objetivo

Organizar um monorepo TS para blockchain com entregas limpas:

- `core`: domínio + tipos + invariantes
- `sdk`: clientes (EVM/BTC/LN/Liquid) com contratos estáveis
- `api`: HTTP/gRPC (DTO + validação)
- `infra`: DB, filas, adapters, docker
- releases sem quebrar contratos (SemVer + compat)
- CI/CD com testes e publish

20.1 Estrutura de pastas

```
repo/
  packages/
    core/
      src/
      package.json
      tsconfig.json
    sdk/
      src/
      package.json
    api/
      src/
      package.json
    infra/
      src/
      package.json
  tsconfig.base.json
  package.json
  pnpm-workspace.yaml
  .github/workflows/ci.yml
```

20.2 Contratos e versionamento (SemVer de verdade)

Definição: Regra

Tudo que é **public API** (export do SDK/core) é contrato. Mudança que quebra tipo = **major**. Adicionar campo opcional = **minor**. Patch = bugfix sem mudar tipos públicos.

Dica prática: Compat sem quebrar

Quando precisar mudar:

- mantenha função antiga por 1-2 versões com `@deprecated`
- crie nova função com nome/assinatura nova
- migre internamente e só depois remova (major)

20.3 Releases automatizadas (conceito)

```
// Use changesets (recomendado)
// pnpm i -D @changesets/cli
// npx changeset init
```

20.4 CI/CD mínimo (checks que importam)

Checklist: Pipeline

- install (pnpm)
- typecheck (tsc -b)
- lint (eslint)
- test (unit + integração opcional por job)
- build (tsup/rollup)
- publish (apenas em tag/release)

```
/*
.github/workflows/ci.yml (pseudo)
- pnpm install
- pnpm -r typecheck
- pnpm -r test
- pnpm -r build
*/
```

21 Parte XX — Implementações de Produção (o que faltava para ser definitivo)

Por que isso importa: Objetivo

Transformar os capítulos (EVM/BTC/LN/Liquid/Arquitetura/Monorepo) em **módulos fechados e implementáveis**: interfaces públicas, adapters, persistência, idempotência, reconciliação e invariantes.

21.1 20.1 Padrão unificado: Core → Adapters → Jobs → Reconciliação

Definição: Regra de ouro (produção)

Tudo que move fundos vira: **Command** (input validado) → **Decision** (domínio) → **Events** (imutáveis) → **Effects** (adapters) → **Reconcile** (source of truth).

```
// core/contracts.ts (public API do core)
export type CommandId = string & { readonly __brand: "CommandId" };
export type IdempotencyKey = string & { readonly __brand: "IdempotencyKey" };

export type CommandMeta = Readonly<{
  commandId: CommandId;
  idem: IdempotencyKey;
  at: string; // ISO
  actor?: string;
}>;

// Um Command SEMPRE  validado na borda (unknown -> parse)
```

```ts
export type WithdrawCommand = Readonly<{
  meta: CommandMeta;
  walletId: string;
  network: "evm" | "btc" | "ln" | "liquid";
  asset: "ETH" | "BTC" | "LBTC" | "USDT" | string;
  amount: bigint; // sempre bigint no core
  destination: string; // address/bolt11
}>;

export type DomainEvent =
  | { kind: "withdraw.requested"; meta: CommandMeta; walletId: string; asset: string; amount:
      bigint; destination: string }
  | { kind: "withdraw.submitted"; meta: CommandMeta; networkRef: string } // txHash/txid/
      paymentHash
  | { kind: "withdraw.settled"; meta: CommandMeta; networkRef: string; confirmations?: number
      }
  | { kind: "withdraw.failed"; meta: CommandMeta; reason: string };

export type WithdrawalState =
  | { status: "none" }
  | { status: "requested"; walletId: string; asset: string; amount: bigint; destination:
      string; idem: IdempotencyKey }
  | { status: "submitted"; walletId: string; asset: string; amount: bigint; destination:
      string; idem: IdempotencyKey; networkRef: string }
  | { status: "settled"; walletId: string; asset: string; amount: bigint; destination: string;
       networkRef: string; confirmations?: number }
  | { status: "failed"; walletId: string; asset: string; amount: bigint; destination: string;
      reason: string };

export function foldWithdrawal(s: WithdrawalState, ev: DomainEvent): WithdrawalState {
  switch (ev.kind) {
    case "withdraw.requested":
      return { status: "requested", walletId: ev.walletId, asset: ev.asset, amount: ev.amount,
          destination: ev.destination, idem: ev.meta.idem };
    case "withdraw.submitted":
      if (s.status !== "requested") return s;
      return { status: "submitted", walletId: s.walletId, asset: s.asset, amount: s.amount,
          destination: s.destination, idem: s.idem, networkRef: ev.networkRef };
    case "withdraw.settled":
      if (s.status !== "submitted") return s;
      return { status: "settled", walletId: s.walletId, asset: s.asset, amount: s.amount,
          destination: s.destination, networkRef: s.networkRef, confirmations: ev.confirmations
           };
    case "withdraw.failed":
      if (s.status === "none") return s;
      return { status: "failed", walletId: s.walletId, asset: s.asset, amount: s.amount,
          destination: s.destination, reason: ev.reason };
  }
}
```

21.2 20.2 EVM (Ethers v6) — TxManager definitivo: retries + nonce + replacement + confirmações

Definição: O que falta em 99% dos códigos EVM

Um **TxManager** que:

- persiste `idem` + `nonce` + `txHash` antes/depois do envio
- faz **replacement** (EIP-1559 bump) quando está stuck
- resiste a **reorg** (N confirmações)
- isola provider (fallback) e concorrência (lock por address)

```ts
// sdk/evm/txManager.ts
import { ethers } from "ethers";

export type Address = string & { readonly __brand: "EvmAddress" };
export type TxHash = string & { readonly __brand: "EvmTxHash" };
export const asAddress = (x: string) => x as Address;

export type FeeBumpPolicy = Readonly<{
  confirmations: number; // ex: 3-12
  bumpPercent: bigint; // ex: 15n
  maxBumps: number; // ex: 5
  bumpAfterMs: number; // ex: 30_000
}>;

export interface TxStore {
  // idem -> tx record (garante idempotncia no submit)
  getByIdem(idem: string): Promise<{ txHash: TxHash; nonce: number } | null>;
  savePlanned(idem: string, from: Address, nonce: number): Promise<void>;
  saveSubmitted(idem: string, txHash: TxHash): Promise<void>;
  markConfirmed(txHash: TxHash, blockNumber: number): Promise<void>;
  markFailed(idem: string, reason: string): Promise<void>;

  // para replacement
  getTxMeta(txHash: TxHash): Promise<{ from: Address; nonce: number; bumps: number;
      lastSubmitAtMs: number } | null>;
  incBumps(txHash: TxHash): Promise<void>;
  touchSubmitTime(txHash: TxHash, atMs: number): Promise<void>;
}

class Mutex {
  private busy = Promise.resolve();
  run<T>(fn: () => Promise<T>): Promise<T> {
    const next = this.busy.then(fn, fn);
    this.busy = next.then(() => undefined, () => undefined);
    return next;
  }
}
const locks = new Map<string, Mutex>();
function lockKey(addr: Address) { return `evm:${addr}`; }
function getLock(addr: Address) {
  const k = lockKey(addr);
  const m = locks.get(k) ?? new Mutex();
  locks.set(k, m);
  return m;
}

export class EvmTxManager {
```

```
constructor(
  private readonly provider: ethers.Provider, // preferir provider resiliente
  private readonly signer: ethers.Signer,
  private readonly store: TxStore,
  private readonly policy: FeeBumpPolicy
) {}

private async feeBump(fee: ethers.FeeData): Promise<{ maxFeePerGas: bigint;
    maxPriorityFeePerGas: bigint }> {
  const bump = (x: bigint) => x + (x * this.policy.bumpPercent) / 100n;
  const maxFee = fee.maxFeePerGas ?? 0n;
  const prio = fee.maxPriorityFeePerGas ?? 0n;
  return { maxFeePerGas: bump(maxFee), maxPriorityFeePerGas: bump(prio) };
}

async submitIdempotent(
  idem: string,
  buildTx: (opts: { nonce: number; fee: { maxFeePerGas: bigint; maxPriorityFeePerGas: bigint
      } }) => Promise<ethers.TransactionResponse>,
): Promise<{ txHash: TxHash; nonce: number }> {
  const from = asAddress(await this.signer.getAddress());

  // lock por address (nonce safety)
  return getLock(from).run(async () => {
    const existing = await this.store.getByIdem(idem);
    if (existing) return existing;

    // nonce "pending" como base
    const nonce = await this.provider.getTransactionCount(from, "pending");
    await this.store.savePlanned(idem, from, nonce);

    // simulao fica dentro do buildTx (callStatic) no contrato
    const fee = await this.provider.getFeeData();
    const bumped = await this.feeBump(fee);

    const tx = await buildTx({ nonce, fee: bumped });
    const hash = tx.hash as TxHash;
    await this.store.saveSubmitted(idem, hash);
    return { txHash: hash, nonce };
  });
}

async waitConfirmations(txHash: TxHash): Promise<void> {
  const receipt = await this.provider.waitForTransaction(txHash, this.policy.confirmations);
  if (!receipt) throw new Error("tx receipt null");
  if (receipt.status === 0) throw new Error("tx reverted");
  await this.store.markConfirmed(txHash, receipt.blockNumber ?? 0);
}

// worker peridico: detecta stuck e faz replacement (mesmo nonce)
async bumpIfStuck(txHash: TxHash): Promise<void> {
  const meta = await this.store.getTxMeta(txHash);
  if (!meta) return;

  if (meta.bumps >= this.policy.maxBumps) return;
  const age = Date.now() - meta.lastSubmitAtMs;
  if (age < this.policy.bumpAfterMs) return;

  // se j confirmou, no faz nada
  const r = await this.provider.getTransactionReceipt(txHash);
```

```
    if (r?.blockNumber) return;

    const fee = await this.provider.getFeeData();
    const bumped = await this.feeBump(fee);

    // replacement: voc precisa reconstruir a tx original (to/data/value/gas)
    // Estratgia prtica: persistir "txRequest" serializvel quando enviar
    // e aqui apenas reenviar com mesmo nonce + fee maior.
    // (Fica como contrato do store; implemente no seu projeto real.)
    await this.store.incBumps(txHash);
    await this.store.touchSubmitTime(txHash, Date.now());
  }
}
```

Dica prática: Dica prática: ABI typing de verdade

O handbook já mostra ABI como `as const`. Para ficar "definitivo" em time grande:

- gere tipos a partir do ABI (TypeChain/abitype) **no SDK**
- mantenha wrapper estável (não exporte o Contract cru)

21.3 20.3 Bitcoin (PSBT) — Engine completo: script templates + coin selection avançado + RBF/CPFP

Definição: O que torna PSBT "produção"

Você separa em 3 camadas:

- **Core**: tipos + políticas + seleção + invariantes
- **Builder**: adapter para `bitcoinjs-lib` (ou outra)
- **Mempool**: monitor + RBF/CPFP decision + reconcile

```
// core/btc/types.ts
export type TxId = string & { readonly __brand: "BtcTxId" };
export type Address = string & { readonly __brand: "BtcAddress" };
export type ScriptPubKey = string & { readonly __brand: "ScriptPubKeyHex" };
export type Sats = bigint & { readonly __brand: "Sats" };
export type FeeRate = bigint & { readonly __brand: "SatPerVb" };
export const sats = (x: bigint) => x as Sats;
export const feeRate = (x: bigint) => x as FeeRate;

export type ScriptType = "p2wpkh" | "p2tr" | "p2sh-p2wpkh" | "unknown";

export type Utxo = Readonly<{
  txid: TxId;
  vout: number;
  value: Sats;
  scriptPubKey: ScriptPubKey;
  scriptType: ScriptType;
  confirmations?: number;
  account?: string;
  isChange?: boolean;
  label?: string;
  // RBF/CPFP hints:
  sequence?: number;
}>;

export type Output = Readonly<{ address: Address; value: Sats }>;
```

```
export type DustPolicy = Readonly<{ minOutput: Sats }>;
export const isDust = (v: Sats, p: DustPolicy) => (v as bigint) < (p.minOutput as bigint);

// vbytes envelopes (ajuste por seu stack)
export function estInVb(t: ScriptType): bigint {
  switch (t) {
    case "p2wpkh": return 68n;
    case "p2tr": return 58n;
    case "p2sh-p2wpkh": return 91n;
    default: return 110n;
  }
}
export const estOutVb = () => 31n;
export const estOverheadVb = () => 10n;

export function estimateFee(inputs: readonly Utxo[], outputsCount: number, fr: FeeRate): Sats
    {
  const vin = inputs.reduce((a,u)=>a+estInVb(u.scriptType), 0n);
  const vout = BigInt(outputsCount) * estOutVb();
  const vb = estOverheadVb() + vin + vout;
  return sats(vb * (fr as bigint));
}
```

```
// core/btc/coinselect.ts
import { Sats, sats, Utxo, Output, FeeRate, estimateFee, DustPolicy, isDust } from "./types";

export type SelectPolicy = Readonly<{
  feeRate: FeeRate;
  dust: DustPolicy;
  minConfs: number;
  maxInputs: number;
  avoidMixingAccounts?: boolean;
  preferFewerInputs?: boolean;
  allowUnconfirmed?: boolean;
}>;

export type SelectError =
  | { kind: "NotEnoughFunds"; have: Sats; need: Sats }
  | { kind: "TooManyInputs"; inputs: number; max: number }
  | { kind: "DustChange"; change: Sats };

export type SelectResult = Readonly<{
  inputs: readonly Utxo[];
  outputs: readonly Output[];
  fee: Sats;
  change?: Output;
  vbytes: bigint;
}>;

const sum = (xs: readonly { value: Sats }[]) => sats(xs.reduce((a,x)=>a + (x.value as bigint),
     0n));
const sumOut = (xs: readonly Output[]) describes => sats(xs.reduce((a,x)=>a + (x.value as
    bigint), 0n));

export function coinSelectAdvanced(
  utxos: readonly Utxo[],
  outputs: readonly Output[],
  changeAddr: Output["address"],
```

```
  pol: SelectPolicy
): SelectResult | SelectError {
  const target = sumOut(outputs);

  let cands = utxos.filter(u => (u.confirmations ?? 0) >= pol.minConfs || !!pol.
      allowUnconfirmed);

  if (pol.avoidMixingAccounts) {
    const acc = cands.find(u=>u.account)?.account;
    if (acc) cands = cands.filter(u=>u.account === acc);
  }

  cands = [...cands].sort((a,b)=>{
    const av = a.value as bigint, bv = b.value as bigint;
    return pol.preferFewerInputs ? Number(bv-av) : Number(av-bv);
  });

  const picked: Utxo[] = [];
  let have = 0n;

  for (const u of cands) {
    picked.push(u);
    have += (u.value as bigint);

    const feeNoChange = estimateFee(picked, outputs.length, pol.feeRate);
    const needNoChange = (target as bigint) + (feeNoChange as bigint);

    if (have >= needNoChange) {
      // tenta change
      const feeWithChange = estimateFee(picked, outputs.length + 1, pol.feeRate);
      const needWithChange = (target as bigint) + (feeWithChange as bigint);
      const change = have - needWithChange;

      if (picked.length > pol.maxInputs) return { kind: "TooManyInputs", inputs: picked.length,
           max: pol.maxInputs };

      if (change <= 0n) {
        const vb = (feeNoChange as bigint) / (pol.feeRate as bigint);
        return { inputs: picked, outputs, fee: feeNoChange, change: undefined, vbytes: vb };
      }

      const changeS = sats(change);
      if (isDust(changeS, pol.dust)) return { kind: "DustChange", change: changeS };

      const vb = (feeWithChange as bigint) / (pol.feeRate as bigint);
      return { inputs: picked, outputs, fee: feeWithChange, change: { address: changeAddr,
          value: changeS }, vbytes: vb };
    }
  }

  const haveS = sats(have);
  const feeGuess = estimateFee(picked, outputs.length + 1, pol.feeRate);
  const needGuess = sats((target as bigint) + (feeGuess as bigint));
  return { kind: "NotEnoughFunds", have: haveS, need: needGuess };
}
```

Por que isso importa: RBF/CPFP como decisão de produto (implementável)

Você não "atira fee" — você decide: RBF se opt-in; CPFP se dá pra gastar um output; senão, reconcilia e espera.

21.4 20.4 Lightning — Máquina de estados completa + idempotência + dedup + reconciliação contínua

Definição: O padrão que evita caos em LN

Event-sourcing leve: eventos do nó (subscribe) + polling periódico (lookup) + dedup por `paymentHash`.

```
// core/ln/model.ts
export type PaymentHash = string & { readonly __brand: "PaymentHash" };
export type Preimage = string & { readonly __brand: "Preimage" };
export type Bolt11 = string & { readonly __brand: "Bolt11" };
export type Msat = bigint & { readonly __brand: "Msat" };
export const msat = (x: bigint) => x as Msat;

export type LnEvent =
  | { kind: "ln.payment.created"; paymentHash: PaymentHash; bolt11: Bolt11; idem: string; at:
      string }
  | { kind: "ln.htlc.offered"; paymentHash: PaymentHash; chanId: string; amount: Msat; at:
      string }
  | { kind: "ln.htlc.settled"; paymentHash: PaymentHash; preimage: Preimage; at: string }
  | { kind: "ln.htlc.failed"; paymentHash: PaymentHash; reason: string; at: string }
  | { kind: "ln.payment.expired"; paymentHash: PaymentHash; at: string };

export type LnState =
  | { status: "none" }
  | { status: "created"; paymentHash: PaymentHash; bolt11: Bolt11; idem: string; at: string }
  | { status: "inflight"; paymentHash: PaymentHash; bolt11: Bolt11; idem: string; lastEventAt:
       string }
  | { status: "succeeded"; paymentHash: PaymentHash; preimage: Preimage; settledAt: string }
  | { status: "failed"; paymentHash: PaymentHash; reason: string; failedAt: string }
  | { status: "expired"; paymentHash: PaymentHash; expiredAt: string };

export function foldLn(s: LnState, ev: LnEvent): LnState {
  switch (ev.kind) {
    case "ln.payment.created": return { status: "created", paymentHash: ev.paymentHash, bolt11:
         ev.bolt11, idem: ev.idem, at: ev.at };
    case "ln.htlc.offered":
      if (s.status === "created" || s.status === "inflight") return { status: "inflight",
          paymentHash: s.paymentHash, bolt11: s.bolt11, idem: s.idem, lastEventAt: ev.at };
      return s;
    case "ln.htlc.settled": return { status: "succeeded", paymentHash: ev.paymentHash,
        preimage: ev.preimage, settledAt: ev.at };
    case "ln.htlc.failed": return { status: "failed", paymentHash: ev.paymentHash, reason: ev.
        reason, failedAt: ev.at };
    case "ln.payment.expired": return { status: "expired", paymentHash: ev.paymentHash,
        expiredAt: ev.at };
  }
}
```

```
// sdk/ln/reconcile.ts
import { LnState, PaymentHash } from "./model";
```

```ts
export interface LightningNodePort {
  lookupPayment(hash: PaymentHash): Promise<
    | { status: "succeeded"; preimage: string; settledAt: string }
    | { status: "failed"; reason: string; failedAt: string }
    | { status: "pending" }
    | { status: "unknown" }
  >;
}

export async function reconcileLn(node: LightningNodePort, state: LnState): Promise<LnState> {
  if (state.status === "none") return state;
  const remote = await node.lookupPayment(state.paymentHash);
  if (remote.status === "succeeded" && state.status !== "succeeded") {
    return { status: "succeeded", paymentHash: state.paymentHash, preimage: remote.preimage as
        any, settledAt: remote.settledAt };
  }
  if (remote.status === "failed" && state.status !== "failed") {
    return { status: "failed", paymentHash: state.paymentHash, reason: remote.reason, failedAt:
        remote.failedAt };
  }
  return state;
}
```

21.5 20.5 Liquid — Política de custódia + auditoria “fechada” (4-olhos, limites, trilha)

Definição: O que é “definitivo” em custódia

Não é só transferir: é **governança codificada**. Cada mudança de estado deve produzir um evento auditável e um lançamento contábil.

```ts
// core/liquid/policy.ts
export type ActorId = string & { readonly __brand: "ActorId" };
export type AssetId = string & { readonly __brand: "AssetId" };
export type Sats = bigint & { readonly __brand: "Sats" };

export type Approval = Readonly<{
  actor: ActorId;
  at: string;
  decision: "approve" | "reject";
  reason?: string;
}>;

export type CustodyPolicy = Readonly<{
  vaultId: string;
  asset: AssetId;
  dailyLimit: Sats;
  requireApprovals: number;
  approvers: readonly ActorId[];
}>;

export type Transfer = Readonly<{
  id: string;
  vaultId: string;
  toAddress: string;
  asset: AssetId;
  amount: Sats;
```

```
  memo?: string;
}>;

export type TransferState =
  | { status: "requested"; tx: Transfer; approvals: readonly Approval[]; requestedAt: string }
  | { status: "approved"; tx: Transfer; approvals: readonly Approval[]; approvedAt: string }
  | { status: "rejected"; tx: Transfer; approvals: readonly Approval[]; rejectedAt: string;
      reason: string }
  | { status: "broadcasted"; tx: Transfer; txid: string; broadcastedAt: string }
  | { status: "settled"; tx: Transfer; txid: string; settledAt: string };

export function applyApproval(s: TransferState, pol: CustodyPolicy, a: Approval):
    TransferState {
  if (s.status !== "requested") return s;
  if (!pol.approvers.includes(a.actor)) return s;

  const approvals = [...s.approvals, a].filter((v,i,arr)=> i === arr.findIndex(x=>x.actor===v.
      actor));
  const rejects = approvals.filter(x=>x.decision==="reject");
  if (rejects.length) return { status:"rejected", tx:s.tx, approvals, rejectedAt:a.at, reason:
       rejects[0].reason ?? "rejected" };

  const approves = approvals.filter(x=>x.decision==="approve");
  if (approves.length >= pol.requireApprovals) return { status:"approved", tx:s.tx, approvals,
       approvedAt:a.at };
  return { ...s, approvals };
}
```

21.6 20.6 Arquitetura — Exactly-once pragmático completo: Outbox + Inbox + IdempotencyKey + Ledger invariants

Por que isso importa: Por que isso fecha o loop

Seu PDF já tem o núcleo Outbox/Inbox/Ledger. Agora fechamos como **pipeline implementável**:

- request entra (idem)
- grava event + outbox (transação)
- worker publica (at-least-once)
- consumer aplica com inbox (dedup)
- reconciliação corrige divergência com nodes
- ledger valida invariantes

```
// infra/outbox/worker.ts
export type OutboxRow = Readonly<{ id: string; topic: string; payloadJson: string; createdAt:
    string; publishedAt?: string }>;

export interface OutboxRepo {
  fetchUnpublished(limit: number): Promise<OutboxRow[]>;
  markPublished(id: string, at: string): Promise<void>;
}

export interface Publisher {
  publish(topic: string, key: string, payloadJson: string): Promise<void>;
}

export async function runOutboxOnce(repo: OutboxRepo, pub: Publisher, limit = 50) {
  const rows = await repo.fetchUnpublished(limit);
```

```
  for (const r of rows) {
    await pub.publish(r.topic, r.id, r.payloadJson);
    await repo.markPublished(r.id, new Date().toISOString());
  }
}
```

```
// infra/inbox/consumeOnce.ts
export interface InboxRepo {
  hasProcessed(messageId: string): Promise<boolean>;
  markProcessed(messageId: string): Promise<void>;
}

export async function consumeOnce(inbox: InboxRepo, messageId: string, handler: () => Promise<
    void>) {
  if (await inbox.hasProcessed(messageId)) return;
  await handler();
  await inbox.markProcessed(messageId);
}
```

```
// core/ledger/invariants.ts
export type LedgerAccount =
  | "customer.available"
  | "customer.pending"
  | "vault.hot"
  | "vault.cold"
  | "fee.revenue"
  | "network.fees";

export type LedgerEntry = Readonly<{
  entryId: string;
  at: string;
  asset: string;
  amount: bigint; // sempre positivo
  debit: LedgerAccount;
  credit: LedgerAccount;
  ref: string; // txId/swapId/paymentHash
}>;

export function assertLedgerEntry(e: LedgerEntry) {
  if (e.amount <= 0n) throw new Error("ledger amount must be > 0");
  if (e.debit === e.credit) throw new Error("debit/credit must differ");
}

// regra simples: cada entry individual balanceada por construo;
// regras reais: somas por ref, limites por poltica, reconciliao com nodes.
```

21.7 20.7 Monorepo + Releases — Setup completo (packages, build, types, CI, compat)

Definição: O que torna monorepo "definitivo"

- **core** sem deps de infra (só tipos + lógica)
- **sdk** encapsula libs (ethers/bitcoinjs/ln grpc)
- **api** faz parse `unknown -> typed` e chama use-cases
- **infra** implementa ports, DB, filas, docker, observabilidade
- releases com **Changesets** + SemVer + deprecations

```
// repo/tsconfig.base.json (sugesto forte)
{
  "compilerOptions": {
    "target": "ES2022",
    "moduleResolution": "node16",
    "module": "Node16",
    "strict": true,
    "declaration": true,
    "declarationMap": true,
    "noUncheckedIndexedAccess": true,
    "exactOptionalPropertyTypes": true,
    "useUnknownInCatchVariables": true,
    "forceConsistentCasingInFileNames": true,
    "noImplicitOverride": true,
    "noFallthroughCasesInSwitch": true,
    "skipLibCheck": true
  }
}
```

```
// packages/core/package.json (exemplo)
{
  "name": "@repo/core",
  "version": "0.1.0",
  "type": "module",
  "exports": {
    ".": {
      "types": "./dist/index.d.ts",
      "default": "./dist/index.js"
    }
  },
  "scripts": {
    "build": "tsc -p tsconfig.json",
    "typecheck": "tsc -p tsconfig.json --noEmit",
    "test": "node --test"
  }
}
```

```
// .github/workflows/ci.yml (pseudo real)
name: ci
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
```

```
      - uses: pnpm/action-setup@v4
      - run: pnpm install --frozen-lockfile
      - run: pnpm -r typecheck
      - run: pnpm -r test
      - run: pnpm -r build
```

Dica prática: Versionamento sem quebrar contratos

Regra prática:

- mudar tipo exportado = **major**
- adicionar campo opcional = **minor**
- bugfix sem mudar tipos públicos = **patch**
- quando precisar mudar: mantenha API antiga 1–2 releases com `@deprecated`

22 Parte XXI — Trade-offs e Quando Não Usar Estes Padrões

22.1 21.1 Engenharia não é aplicar padrão — é escolher custo

Por que isso importa: Engenharia é escolha

Os padrões apresentados neste handbook existem para reduzir risco em sistemas que movem dinheiro ou estado irreversível. Eles têm custo real:

- mais tipos
- mais código
- mais conceitos
- onboarding mais lento
- maior carga cognitiva

Nem todo sistema precisa disso.

Definição: A pergunta correta

A pergunta correta não é: *"Qual é o padrão mais correto?"*
É: **"Qual é o menor nível de complexidade que mantém o sistema seguro?"**

22.2 21.2 Quando NÃO usar Result<T,E>

Definição: Regra de decisão

Use **exceções** quando:

- o erro é inesperado
- indica bug ou estado impossível
- não existe ação de domínio para tratar

Use **Result** quando:

- o erro faz parte do fluxo normal
- o usuário pode causar o erro
- retry depende do tipo de erro
- o sistema precisa reagir

```
// Exceo: erro inesperado (bug / invariantes)
throw new Error("invariant broken");

// Result: erro esperado (domnio)
type DomainError =
  | { kind: "InsufficientFunds"; balance: bigint; needed: bigint }
  | { kind: "InvalidAddress"; reason: string }
  | { kind: "RateLimited"; retryAfterMs?: number }
  | { kind: "RpcTimeout"; provider: string };
```

Dica prática: Tabela mental (cole na cabeça)

Situação	Estratégia
erro esperado	Result
erro inesperado	exception

22.3 21.3 Custo cognitivo de tipos excessivos

Por que isso importa: Tipos têm retorno decrescente

Tipos aumentam segurança, mas também:

- aumentam fricção
- tornam refactors mais caros
- dificultam leitura para novos devs

O objetivo não é "tipar tudo" — é **impedir erros caros**.

Erro comum: Anti-padrão comum

Brand types em um CRUD simples onde IDs nunca se misturam:

```
type UserId = string & { readonly __brand: "UserId" };
type WalletId = string & { readonly __brand: "WalletId" };
type SessionId = string & { readonly __brand: "SessionId" };
```

Dica prática: Quando brand types valem a pena

Use brand types quando:

- misturar IDs causa perda financeira
- o sistema tem múltiplos agregados (swapId vs txId vs paymentHash)
- bugs seriam silenciosos (string → string)

Não use quando:

- o domínio é simples e o time é pequeno
- a chance de mistura é baixa e o custo de bug é pequeno

22.4 21.4 Quando simplificar

Definição: Simplificar é maturidade

Você pode simplificar quando:

- sistema é interno
- dados são descartáveis
- não há movimento de fundos
- inconsistência é aceitável
- rollback é barato

```
// Exemplo vlido (sistema simples, custo de erro baixo)
async function getUserBalance(id: string): Promise<number> {
  return 42;
}
```

Por que isso importa: Engenharia madura sabe onde parar

Não precisa de Result, BigInt, branding, state machine para todo endpoint. Você escolhe o menor conjunto que mantém o sistema seguro.

23 Parte XXII — Observabilidade (Onde Sistemas Reais São Debugados)

23.1 22.1 A verdade dura

Definição: Produção é investigação

Em produção: você passa mais tempo investigando sistemas do que escrevendo código.
Logs sem contexto são inúteis.

23.2 22.2 Correlation ID (regra obrigatória)

Por que isso importa: Por que isso é obrigatório

Toda operação que atravessa serviços deve possuir um **CorrelationId** e ele deve ser propagado em:

- logs
- eventos
- chamadas RPC
- filas

Sem isso, debugging distribuído é praticamente impossível.

```
type CorrelationId = string & { readonly __brand: "CorrelationId" };

type LogContext = Readonly<{
  correlationId: CorrelationId;
  swapId?: string;
  walletId?: string;
  txHash?: string;
  paymentHash?: string;
}>;

function logInfo(msg: string, ctx: LogContext) {
  // seu logger real: pino/winston/otel
  console.log(JSON.stringify({ level: "info", msg, ...ctx }));
}
```

Dica prática: Formato simples que funciona

Exemplo de campos que salvam sua vida:

- `correlationId=req-98f2`
- `swapId=abc123`
- `txHash=0x...`
- `paymentHash=...`

23.3 22.3 Tracing (OpenTelemetry mental model)

Definição: Spans como anatomia de operação

Cada operação longa deve ser quebrada em spans:

```
// swap.execute
//  quote.fetch
```

```
//  utxo.select
//  tx.build
//  broadcast
//  reconcile
```

Por que isso importa: O objetivo não é "métricas bonitas"

O objetivo é responder rápido:

- Onde o sistema está lento?
- Onde está quebrando?
- Qual dependência externa está falhando?
- Quantos retries estão acontecendo?

23.4 22.4 Métricas que realmente importam

Definição: Não medir tudo

Meça o que indica **risco** e **correção de estado**.

Checklist: Conjunto mínimo

Wallet / Swap

- `withdrawals_started`
- `withdrawals_failed`
- `retries_total`
- `reconciliation_mismatches`

Lightning

- `payments_inflight`
- `settle_time_ms`
- `failures_by_reason`

Bitcoin

- `broadcast_latency`
- `mempool_stuck_count`

Erro comum: Frase para lembrar

Se você não mede reconciliação, você não sabe se perdeu dinheiro.

24 Parte XXIII — Falhas Reais (O Que Quebra em Produção)

Por que isso importa: Por que esta seção existe

Esta seção separa handbook acadêmico de handbook vivo.
Aqui estão falhas reais que acontecem mesmo quando "o código parece correto".

24.1 23.1 Double spend lógico (sem blockchain)

Definição: Cenário real

`withdraw request` → retry por timeout → segundo job executa → dois broadcasts.

Erro comum: Diagnóstico

A blockchain não errou. O backend errou.
Causa:

- ausência de `idempotencyKey`
- estado não persistido antes do broadcast

Dica prática: Correção

- persistir estado antes do efeito externo
- operação idempotente por `idempotencyKey`
- dedup em consumer (Inbox)

24.2 23.2 Retry sem idempotência

Erro comum: Erro clássico

```
// ERRO: se sendTransaction no for idempotente -> mltiplas transaes
await retry(() => sendTransaction());
```

Definição: Regra

Retry só é seguro quando o efeito externo está protegido por chave idempotente (ou quando o efeito é naturalmente idempotente).

24.3 23.3 Reorg quebrando estado

Definição: Erro comum

Evento detectado → saldo atualizado → bloco reorgado → evento desaparece.

Dica prática: Correção

- nunca tratar evento como final
- exigir **N confirmações**
- reconciliação periódica

24.4 23.4 Lightning settle duplicado (estado local divergente)

Definição: Cenário

Node LN confirma pagamento, mas:

- evento websocket é perdido
- estado local continua `pending`

Dica prática: Correção

Reconciliação periódica com o node LN (source of truth).

Por que isso importa: Conclusão

Os padrões do handbook (idempotência, reconciliação, estados explícitos, outbox/inbox) existem porque **o mundo real é assíncrono e falha**.

25 Parte XXIV — Operational Philosophy (Doutrina Operacional)

Por que isso importa: Por que esta parte muda tudo

O que falta para um handbook virar nível exchange/custodian/infra séria não é mais técnica — é **doutrina operacional**: como decisões são tomadas sob pressão.

25.1 24.1 A premissa fundamental

Definição: Risco real

Em sistemas financeiros, o maior risco não é erro visível.
É **erro silencioso**.

Por que isso importa: Prioridade correta

1. **Correção do estado**
2. **Auditabilidade**
3. **Recuperabilidade**
4. Performance
5. Conveniência

Nunca o contrário.

25.2 24.2 Falhas aceitáveis vs inaceitáveis

Definição: Falhas aceitáveis

- request falhar
- retry necessário
- latência maior
- serviço temporariamente indisponível

O usuário pode tentar novamente.

Erro comum: Falhas inaceitáveis

- perda de fundos
- estado contábil incorreto
- divergência entre ledger e blockchain
- operação executada duas vezes
- sucesso reportado sem efeito real

Essas falhas justificam: idempotência, reconciliação, validação runtime, estados explícitos.

25.3 24.3 Source of Truth (regra absoluta)

Definição: A API nunca é verdade

A API é só uma visão. Todo sistema deve declarar explicitamente sua source of truth.

Dica prática: Tabela de Source of Truth

Componente	Source of Truth
BTC balance	blockchain
LN payment	node LN
swap state	ledger interno
API response	nunca

Por que isso importa: Regra de divergência

Quando há divergência: **source of truth vence** e estado local corrige.

25.4 24.4 Sistemas são eventualmente corrigidos

Definição: Mudança de mentalidade

Mentalidade júnior: o sistema não pode errar.
Mentalidade sênior: o sistema vai errar; ele precisa se corrigir.

Dica prática: Ferramentas de correção

- reconciliação
- event sourcing / rebuild de estado
- reprocessamento
- materialized views reconstruíveis

O objetivo não é perfeição. É **convergência para o estado correto**.

25.5 24.5 Engenharia orientada a recuperação

Definição: Pergunta obrigatória

Toda operação crítica deve responder: **Como recupero isso se falhar no meio?**

```
// Exemplo de cenrio: TX construda -> broadcast falha -> processo reinicia
// Perguntas obrigatrias:
// - o tx hex foi persistido?
// - o nonce foi salvo?
// - o job pode rodar novamente com idempotency?
```

25.6 24.6 O princípio mais importante (nível infra)

Princípio operacional (guarde isso)

Nunca execute um efeito externo antes de persistir o estado que permite recuperá-lo.

Por que isso importa: O que esse princípio evita

- double broadcast
- pagamentos duplicados
- inconsistência após crash
- sucesso "mentiroso" (200 OK sem efeito real)

26 Parte XXV — Failure Budget & Risk Model (Quanto risco o sistema pode aceitar antes de parar)

26.1 25.1 Premissa fundamental

Definição: Nenhum sistema é perfeito

Nodes caem. RPCs falham. Blocos reorganizam. Filas atrasam. Serviços externos mentem.
O objetivo não é eliminar falhas.
O objetivo é **limitar o impacto das falhas**.

Por que isso importa: Por que Failure Budget existe

Sem um modelo explícito de risco, o sistema pode continuar operando enquanto já está incorreto.
Failure Budget define **quanto erro o sistema tolera** antes de reduzir risco automaticamente.

26.2 25.2 O que é Failure Budget

Definição: Definição

Failure budget é o **limite aceitável de falha** dentro de uma janela de tempo. Quando o limite é ultrapassado, o sistema muda de comportamento automaticamente.

Dica prática: Exemplo de budget (janela curta)

Métrica	Limite (exemplo)
`withdrawals_failed`	$< 1\%$ em 10 min
`reconciliation_mismatch`	0 permitido
`broadcast_timeout`	$< 5\%$ em 10 min
`ln_payments_stuck`	$< 2\%$ em 10 min

Por que isso importa: Ação automática

Quando o budget estoura, o sistema entra em modo de risco menor (degraded/safe-mode/halted) para evitar cascata.

26.3 25.3 Classes de falha (peso importa)

Definição: Nem toda falha tem o mesmo peso

O sistema deve classificar falhas por impacto e decidir a resposta por classe.

Classe A — Falha operacional (aceitável)

Exemplos:

- RPC timeout
- retry necessário
- rate limit
- fila atrasada

Ação:

- retry automático
- o sistema continua operando

Classe B — Falha degradada (risco crescente)

Exemplos:

- aumento anormal de retries
- latência elevada sustentada
- falhas acima do baseline

Ação:

- reduzir throughput
- aumentar confirmações
- desabilitar features não críticas

Exemplo operacional: *swaps continuam, withdrawals são pausados temporariamente.*

Classe C — Falha crítica (inaceitável)

Exemplos:

- divergência contábil
- saldo inconsistente
- reconciliação falhando
- efeitos duplicados detectados

Ação:

- parar operações automaticamente
- nenhuma saída de fundos deve continuar

26.4 25.4 Modos operacionais do sistema (estado explícito)

Definição: Estado do sistema é contrato

O sistema deve possuir estados explícitos e auditáveis.

```
export type SystemMode =
  | "normal"
  | "degraded"
  | "safe-mode"
  | "halted";
```

Dica prática: Semântica dos modos

- **NORMAL**: operação completa (todos fluxos ativos)
- **DEGRADED**: dependências instáveis; reduzir risco mantendo operação parcial
- **SAFE-MODE**: inconsistência detectada; bloquear efeitos externos e priorizar correção
- **HALTED**: risco financeiro ativo/estado desconhecido; parar tudo e exigir humano

Checklist: Ações típicas por modo

DEGRADED

- reduzir paralelismo
- aumentar timeout
- bloquear novos swaps grandes
- exigir mais confirmações

SAFE-MODE

- bloquear withdrawals
- permitir apenas leitura
- permitir reconciliação e rebuild de estado

HALTED

- parar todos efeitos externos
- exigir intervenção humana (runbook/incident)

26.5 25.5 Circuit Breakers (proteção automática)

Definição: Objetivo

Impedir cascata de falhas: quando uma dependência fica instável, o sistema muda de modo automaticamente.

```
type Window = Readonly<{ seconds: number }>;

type Threshold = Readonly<{
  window: Window;
  maxRate: number; // ex: 0.01 -> 1%
  maxCount?: number; // opcional
}>;

type MetricName =
  | "withdrawals_failed"
  | "broadcast_timeout"
  | "reconciliation_mismatch"
  | "ln_payments_stuck"
  | "retries_total";

type MetricSample = Readonly<{ atMs: number; ok: boolean }>;

class SlidingWindow {
  private samples: MetricSample[] = [];
  constructor(private readonly windowSec: number) {}

  add(s: MetricSample) {
    this.samples.push(s);
    this.gc();
  }
```

```
  private gc() {
    const min = Date.now() - this.windowSec * 1000;
    while (this.samples.length && this.samples[0].atMs < min) this.samples.shift();
  }

  rateFail(): number {
    this.gc();
    if (!this.samples.length) return 0;
    const fails = this.samples.filter(s => !s.ok).length;
    return fails / this.samples.length;
  }

  countFail(): number {
    this.gc();
    return this.samples.filter(s => !s.ok).length;
  }

  countAll(): number {
    this.gc();
    return this.samples.length;
  }
}

type FailureBudgetPolicy = Readonly<{
  thresholds: Record<MetricName, Threshold>;
}>;

type ModeTransition = Readonly<{
  from: SystemMode;
  to: SystemMode;
  reason: string;
  at: string;
}>;

class RiskEngine {
  private mode: SystemMode = "normal";
  private windows: Map<MetricName, SlidingWindow> = new Map();
  private transitions: ModeTransition[] = [];

  constructor(private readonly policy: FailureBudgetPolicy) {
    for (const k of Object.keys(policy.thresholds) as MetricName[]) {
      const w = policy.thresholds[k].window.seconds;
      this.windows.set(k, new SlidingWindow(w));
    }
  }

  getMode(): SystemMode { return this.mode; }
  getTransitions(): readonly ModeTransition[] { return this.transitions; }

  record(metric: MetricName, ok: boolean) {
    const win = this.windows.get(metric)!;
    win.add({ atMs: Date.now(), ok });
    this.evaluate(metric);
  }

  private setMode(to: SystemMode, reason: string) {
    if (this.mode === to) return;
    this.transitions.push({ from: this.mode, to, reason, at: new Date().toISOString() });
    this.mode = to;
```

```
  }

  private evaluate(metric: MetricName) {
    const th = this.policy.thresholds[metric];
    const win = this.windows.get(metric)!;

    // classe C: mismatch  zero-tolerance
    if (metric === "reconciliation_mismatch") {
      if (win.countFail() > 0) {
        this.setMode("safe-mode", "reconciliation mismatch detected (zero tolerance)");
      }
      return;
    }

    const failRate = win.rateFail();
    const failCount = win.countFail();

    // exemplo: degrade quando passa limite
    if (failRate > th.maxRate || (th.maxCount !== undefined && failCount > th.maxCount)) {
      // certas mtricas podem ir direto para safe-mode dependendo do domnio
      if (metric === "withdrawals_failed") {
        this.setMode("degraded", "withdrawals failing above budget");
      } else if (metric === "broadcast_timeout") {
        this.setMode("degraded", "broadcast timeouts above budget");
      } else if (metric === "ln_payments_stuck") {
        this.setMode("degraded", "LN stuck above budget");
      } else {
        this.setMode("degraded", `${metric} above budget`);
      }
    }
  }

  // chamada por operador/humano
  halt(reason: string) {
    this.setMode("halted", reason);
  }
}
```

Dica prática: Uso prático

- Cada operação crítica registra sucesso/falha em métricas
- O RiskEngine recalcula e ajusta o modo
- Use o modo para **bloquear** efeitos externos (withdraw/broadcast/pay)

26.6 25.6 Quando exigir intervenção humana

Definição: Automação tem limite

Nem tudo deve ser automático. Intervenção humana é obrigatória quando:

- **ledger ≠ blockchain**
- LN node reporta estado impossível
- múltiplos retries falham sem causa clara
- comportamento fora do modelo esperado

Motivo: automação sem compreensão **amplifica erro**.

26.7 25.7 Regra de ouro (nível exchange)

Regra de ouro (nível exchange)

É melhor parar o sistema corretamente do que continuar operando incorretamente.
Downtime é recuperável. Perda financeira não é.

26.8 25.8 Integração com reconciliação contínua

Definição: Budget depende de reconciliação

Failure budget só funciona se a reconciliação estiver ativa e contínua.

```
type ReconcileCheck =
  | { kind: "btc.balance"; ok: boolean; details?: string }
  | { kind: "ln.state"; ok: boolean; details?: string }
  | { kind: "evm.nonce"; ok: boolean; details?: string }
  | { kind: "ledger.integrity"; ok: boolean; details?: string };

async function reconcileLoop(risk: RiskEngine) {
  // pseudo: rode a cada X segundos
  const checks: ReconcileCheck[] = [
    { kind: "btc.balance", ok: true },
    { kind: "ln.state", ok: true },
    { kind: "ledger.integrity", ok: true }
  ];

  for (const c of checks) {
    if (!c.ok) {
      // zero tolerance
      risk.record("reconciliation_mismatch", false);
      return;
    }
  }

  risk.record("reconciliation_mismatch", true);
}
```

Por que isso importa: Regra operacional

Se reconciliação detecta mismatch: **safe-mode**.
O sistema prioriza correção, não operação.

26.9 20.8 Checklists finais (produção real)

- **EVM**: idem + nonce lock + replacement + N conf + reorg safe (persistir log keys)
- **BTC**: coin selection com dust + fee/vb + coin control + RBF/CPFP decision
- **LN**: state machine + dedup + reconcile loop + idempotencyKey
- **Liquid**: approvals + limits + audit trail + ledger entries
- **Arquitetura**: event store + outbox/inbox + reconcile com nodes
- **Monorepo**: core/sdks/api/infra com exports estáveis + changesets + CI

Checklist: Checklist final (nível produção real)

- core não depende de infra
- SDK encapsula ethers/bitcoin libs (adapters)
- DTO parse em API é `unknown -> validated`
- idempotência em endpoints que movem fundos
- outbox+inbox em integrações assíncronas
- reconciliação periódica com nodes (EVM/BTC/LN)

27 Links oficiais (source of truth)

- Índice PT: https://www.typescriptlang.org/pt/docs/handbook/
- Intro: https://www.typescriptlang.org/docs/handbook/intro.html
- Narrowing: https://www.typescriptlang.org/docs/handbook/2/narrowing.html
- Generics: https://www.typescriptlang.org/docs/handbook/2/generics.html
- Utility Types: https://www.typescriptlang.org/docs/handbook/utility-types.html