

AlgoPay — Technical Deep Dive

Algorand-powered programmable QR payments & NFT receipts.

Focus: complexities of the stack, how Algorand enables the UX, and flowcharts + pseudocode for each smart-contract path.

1. Technology Complexities & Design Choices

Algorand (Protocol & AVM)

- Pure Proof-of-Stake with VRF-based committees \Rightarrow deterministic finality in $<5s$; UX can rely on consistent timings.
- AVM is a stack-based interpreter (TEAL) with fixed budgets \Rightarrow predictable execution cost, simpler fee modeling.
- Atomic Transaction Groups (≤ 16 txns) \Rightarrow DB-style all-or-nothing across payments/app-calls; removes partial states.
- Stateless TEAL (LogicSig) & Stateful Apps split \Rightarrow custody-as-code + persistent marketplace logic.
- Inner transactions \Rightarrow refunds/receipts can be dispatched by the app without off-chain signers.

PyTeal (Smart Contract Authoring)

- High-level Python DSL compiling to TEAL \Rightarrow safer expressions, but requires careful encoding/decoding of state blobs.
- Deterministic schemas \Rightarrow on-chain K/V constraints; upgrades require migration planning and versioning keys.

LogicSig Escrows

- No private keys; spending rules embedded in code \Rightarrow great security, but requires strict group validation to prevent misuse.
- Deterministic address derivation (app_id+listing_id) \Rightarrow simple lookup, but collisions must be theoretically excluded.

Atomic Groups

- Group construction & signature ordering is sensitive \Rightarrow backend must serialize/group correctly; any mismatch rejects whole group.

Indexer & Node Access

- Event-driven reconciliation via Indexer \Rightarrow simpler than custom logs, but requires consistency handling for catch-up and paging.

Backend (Node.js Orchestrator)

- Transaction builder & signer pipelines must be idempotent; retries must not double-spend due to atomicity guarantees.
- Concurrency: avoid race conditions when multiple buyers target the same listing; use app state checks + optimistic locking.

Frontend (React PWA)

- Camera permissions, QR decoding fallbacks, offline UX for vouchers; strict a11y (WCAG 2.1 AA) impacts focus order and ARIA.

IPFS / Metadata

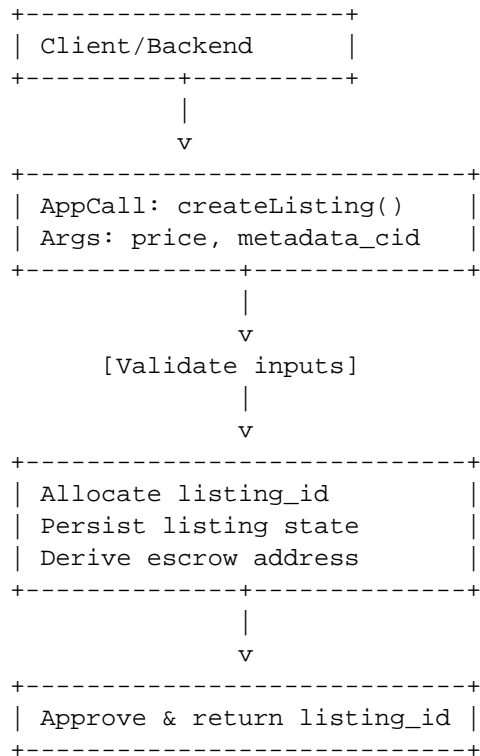
- CID pinning & availability; treat IPFS as content-addressed, immutable source \Rightarrow validate formats and enforce size limits.

2. Smart Contracts Overview

Stateful Marketplace App (PyTeal) + Stateless LogicSig Escrows per listing.

Core flows: createListing → lockPayment (atomic) → finalizePayment (escrow release) → optional receipt NFT; refundPayment (timeout/admin/seller).

2.1 createListing — Flowchart

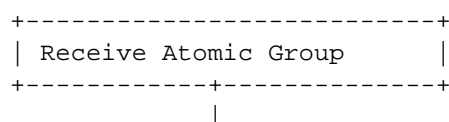


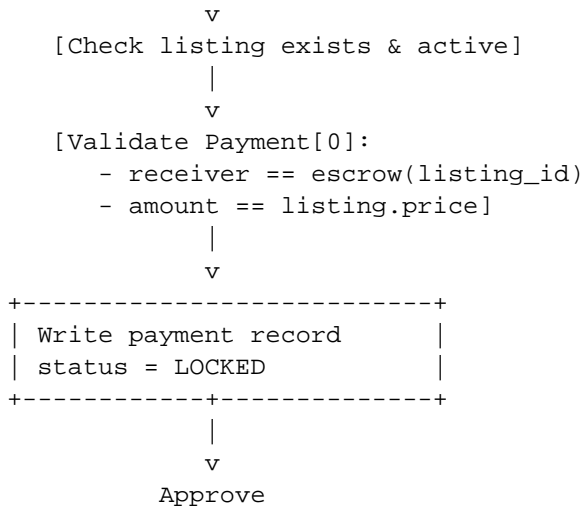
Pseudocode — createListing

```
@router.method
def createListing(price: Uint64, metadata_cid: String, payment: PaymentTxn) -> Uint64:
    Assert(price > 0)
    Assert(is_valid_cid(metadata_cid))
    Assert(payment.amount >= LISTING_FEE)
    listing_id = App.globalGet(Bytes("listing_counter"))
    App.globalPut(listing_key(listing_id), encode_listing(
        creator=Txn.sender(),
        price=price,
        metadata_cid=metadata_cid,
        status=ACTIVE,
        created_at=Global.latest_timestamp()
    ))
    create_escrow_for(listing_id) # deterministic LogicSig parameters
    App.globalPut(Bytes("listing_counter"), listing_id + Int(1))
    return listing_id
```

2.2 lockPayment — Flowchart (Atomic Group with Payment)

Group[0]: Payment Buyer -> Escrow(listing_id)
Group[1]: AppCall lockPayment(listing_id, ref Payment[0])





Pseudocode — lockPayment

```

@router.method
def lockPayment(listing_id: Uint64, payment: PaymentTxn) -> None:
    listing = App.globalGet(listing_key(listing_id))
    Assert(listing != Bytes(""))
    price = decode_listing(listing).price
    escrow = App.globalGet(escrow_key(listing_id))
    Assert(payment.receiver == escrow)
    Assert(payment.amount == price)
    payment_id = Sha256(Concat(Txn.sender(), Itob(listing_id), Itob(Global.latest_timestamp())))
    App.globalPut(payment_id, encode_payment(
        buyer=Txn.sender(),
        listing_id=listing_id,
        amount=price,
        status=LOCKED,
        locked_at=Global.latest_timestamp()
    ))
    Approve()

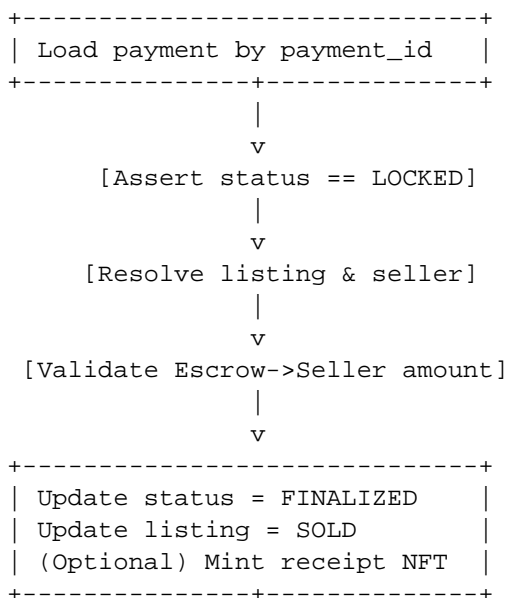
```

2.3 finalizePayment — Flowchart (Release Escrow + Receipt)

```

Group[0]: AppCall finalizePayment(payment_id)
Group[1]: Payment Escrow -> Seller (LogicSig)
Group[2]: (Optional) AssetTransfer ReceiptNFT -> Buyer

```



```

    |
    v
Approve

```

Pseudocode — finalizePayment

```

@router.method
def finalizePayment(payment_id: DynamicBytes, escrow_payment: PaymentTxn) -> None:
    p = App.globalGet(payment_id.get())
    Assert(p != Bytes(""))
    Assert(decode_payment(p).status == LOCKED)
    listing_id = decode_payment(p).listing_id
    l = App.globalGet(listing_key(listing_id))
    seller = decode_listing(l).creator
    Assert(escrow_payment.receiver == seller)
    Assert(escrow_payment.amount == decode_payment(p).amount)
    App.globalPut(payment_id.get(), set_status(p, FINALIZED))
    App.globalPut(listing_key(listing_id), set_listing_status(l, SOLD))
    maybe_mint_receipt(payment_id.get())
    Approve()

```

2.4 refundPayment — Flowchart (Timeout / Admin / Seller)

```

+-----+
| AppCall refundPayment(pid, r) |
+-----+
    |
    v
  [Load & Validate payment]
    |
    v
[Authorize: buyer-after-timeout
  OR admin
  OR seller-before-timeout]
    |
    v
+-----+
| Update status = REFUNDED      |
| InnerTxn: Payment -> Buyer    |
+-----+
    |
    v
  Approve

```

Pseudocode — refundPayment

```

@router.method
def refundPayment(payment_id: DynamicBytes, reason: String) -> None:
    p = App.globalGet(payment_id.get())
    Assert(p != Bytes(""))
    buyer = decode_payment(p).buyer
    listing_id = decode_payment(p).listing_id
    l = App.globalGet(listing_key(listing_id))
    seller = decode_listing(l).creator
    ok = Or(
        And(Txn.sender() == buyer,
            Global.latest_timestamp() > decode_payment(p).locked_at + TIMEOUT),
        Txn.sender() == App.globalGet(Bytes("admin")),
        And(Txn.sender() == seller,
            Global.latest_timestamp() < decode_payment(p).locked_at + TIMEOUT)
    )
    Assert(ok)

```

```

App.globalPut(payment_id.get(), set_status(p, REFUNDED))
InnerTxnBuilder.Begin()
InnerTxnBuilder.SetFields({
    TxnField.type_enum: TxnType.Payment,
    TxnField.receiver: buyer,
    TxnField.amount: decode_payment(p).amount,
    TxnField.fee: Int(0)
})
InnerTxnBuilder.Submit()
Approve()

```

2.5 LogicSig Escrow — Flowchart & Pseudocode

Flow (must be in a group):

```

[Prev Tx] AppCall (finalize/refund, listing_id)
[This Tx] Payment Escrow -> (Seller or Buyer)

```

Guards:

```

- Txn.type == Payment
- Gtxn[group_index-1].application_id == app_id
- Gtxn[group_index-1].args[0] in {finalize, refund}
- Gtxn[group_index-1].args[1] == listing_id

```

LogicSig Pseudocode

```

def escrow_logicsig(app_id: int, listing_id: int):
    Assert(Global.group_size() > Int(1))
    Assert(Txn.type_enum() == TxnType.Payment)
    prev = Gtxn[Txn.group_index() - Int(1)]
    Assert(prev.application_id() == Int(app_id))
    op = prev.application_args[0]
    Assert(Or(op == Bytes("finalize"), op == Bytes("refund")))
    Assert(Btoi(prev.application_args[1]) == Int(listing_id))
    Approve()

```

3. Atomic Group Construction (Backend)

The Node.js orchestrator uses `algosdk` to compose, sign, and submit groups. Ordering and signatures are critical; any mismatch invalidates the whole group.

```
// Pseudocode (Node.js)
const pmt = makePaymentTxn({ from: buyer, to: escrow, amount: price });
const app = makeAppCallTxn({ appId, method: 'lockPayment', args: [listing_id], ref: pmt });
const group = assignGroupId([pmt, app]);
signWithBuyer(pmt);
signWithBuyer(app);
submit(group);
await waitForConfirmation(txid);
```

4. Edge Cases & Failure Modes

- Underpayment/overpayment \Rightarrow `lockPayment` assert fails \Rightarrow whole group rejected.
- Wrong escrow address \Rightarrow assert fails; prevents spoofing.
- Double finalize \Rightarrow status check prevents re-release; idempotent safety.
- Listing paused/sold \Rightarrow pre-check in lock step; prevents new locks.
- Indexer lag \Rightarrow backend waits for confirmed round; UI shows pending until $\text{round} \geq \text{txn.confirmedRound}$.
- Timeouts \Rightarrow `refundPayment` path guarded by timestamps in on-chain state.

5. UX from Protocol Guarantees

- $< 5\text{s}$ deterministic finality \Rightarrow predictable voice/visual feedback and accessible progress states.
- Fixed 0.001 ALGO fees \Rightarrow micro-payments viable; users never encounter price spikes.
- Forkless consensus \Rightarrow no rollbacks; receipts stay valid; explorer links are stable.