identiPay – The Intent-Based Commerce Protocol

from pull payments to contextual, intent-driven atomic settlement.

Krum Sultov

February 23, 2026

Abstract

Almost all modern payments infrastructure is built around *pull payments*: payers disclose static credentials (e.g., PAN/CVV, IBAN, mandate identifiers) to merchants, who can initiate withdrawals subject to external policy and legal constraints. This model is difficult to secure and hard to interpret, fragments identity, increases fraud risk, and often relies on dispute resolution after the fact.

identiPay introduces a *push-only, intent-based* commerce protocol that combines stealth meta-addresses—human-readable names backed by zero-knowledge proofs of government-grade credentials (EUDI-aligned Verifiable Credentials)—with Programmable Transaction Blocks (PTBs). Names resolve to cryptographic public keys, never to blockchain addresses; every payment is delivered to a fresh, one-time stealth address that only the recipient can detect and spend from. Instead of simple value transfer, identiPay executes *contextual atomic settlement*: money, identity proofs, and commerce artifacts (receipts, invoices, warranties) are exchanged in a single, indivisible execution. The same signed intent can be fulfilled via multiple settlement rails (native Sui execution, bank rails, or cross-chain liquidity), without changing the semantics of the commerce agreement.

Contents

1 Introduction

Payments are a security boundary. In legacy rails, that boundary is misplaced: the payer must broadcast an identifier *sufficient to extract value* to an untrusted counterparty. In Web3, the boundary is also misaligned: the payer often signs data without a clear, verifiable binding to human intent. In both cases, the payment rail is disconnected from the commerce rail, creating a reconciliation gap that grows with transaction volume.

identiPay defines a protocol boundary where (i) the merchant can only *propose*, never *charge*; (ii) the payer signs a *semantically typed intent*; and (iii) settlement of value and commerce data is atomic by construction.

2 What Breaks in Today's Payment Flows

2.1 Pull Payments Are a Liability

Structural flaw. In pull systems (cards, direct debits, tokenized card vaulting), the payer provides a credential that authorizes third-party initiation of transfers. In practice, the credential is a long-lived capability: it can be replayed, stolen, or used beyond the payer's intended context.

Consequence. Fraud, subscription traps, and credential compromise are recurring risks in systems that delegate withdrawal authority to many merchants and intermediaries. The payer often lacks a technical "stop" primitive; solutions tend to be legal and after the fact.

2.2 Users Are Asked to Sign What They Can't Read

Structural flaw. Many Web3 wallets still require users to approve transactions described in technical terms (e.g., function selectors, byte arrays, object IDs), and the mapping to user intent is not always clear or verifiable.

Consequence. Users routinely authorize actions they cannot reason about: approvals, delegate capabilities, and transfers that are semantically unrelated to the perceived action (e.g., "buying a laptop"). This is a usability failure that becomes a security failure.

2.3 Payments Lose Context

Structural flaw. Commerce requires more than moving value. Receipts, warranties, invoices, tax metadata, and proof of ownership are typically generated off-ledger and stored on disconnected systems. Value transfer is recorded, while the semantics of the exchange are not.

Consequence. Consumers lose enforceability (lost receipts, unenforceable warranties), and enterprises incur reconciliation costs. A bank statement line item encodes "-$100" but not the verifiable who/what/why of the transaction.

3 Protocol Overview

identiPay combines three core components:

1. **Stealth Identity:** a human-readable name backed by a stealth meta-address—a pair of cryptographic public keys derived from government-grade credentials. Every incoming payment uses a fresh, unlinkable stealth address. Eligibility checks are performed via ZK proofs without exposing PII.

2. **Push-Only Intent Engine:** a typed proposal/acceptance model where merchants propose, wallets accept, and the on-chain execution is constrained by the signed intent.

3. **Atomic Contextual Settlement:** PTB-based execution where value transfer and commerce artifacts exchange atomically.

The protocol is intentionally *deterministic*: the same intent and the same on-chain state yield the same execution; deviations fail safely.

4 identiPay as a Settlement Abstraction Layer

The historical design of payment networks (e.g., Visa, SWIFT, and even monolithic blockchains) tightly couples three distinct functions: identity verification, transaction authorization, and value settlement. If a merchant wishes to accept a new form of value, they must integrate an entirely new parallel stack.

identiPay breaks this monolith by serving as an **Abstraction Layer** over settlement rails. It shifts the paradigm from "Pushing Money" to "Broadcasting Intent."

4.1 Separating Authorization from Settlement

In the identiPay protocol, the core unit of commerce is the **Universal Intent Object (UIO)**. The UIO is a cryptographically signed JSON-LD payload that mathematically binds the payer's stealth identity and any required ZK-proofs to a specific set of commercial terms (e.g., "Transfer 50 EUR value to Merchant DID in exchange for Warranty Object, deliver artifacts to stealth address A_s").

Crucially, the UIO is *rail-agnostic*. The protocol defines how the agreement is negotiated and signed, but does not strictly dictate the underlying pipes used to move the liquidity. identiPay acts as the "Universal Remote," allowing the exact same identity and authorization flow to control disparate settlement engines.

4.2 Cross-Rail Interoperability

identiPay is introduced in phases. The initial deployment is intentionally narrow: **USDC settlement on Sui** with full PTB-level atomicity. Cross-rail settlement is added later via an identiPay **Solver Network** that can fulfill the same signed intent using external liquidity and rails, while the merchant gets their money without having to integrate every source rail the payer might use.

1. **Phase 1: Native Sui Settlement (USDC-first).** The signed UIO is wrapped into a Programmable Transaction Block (PTB) and executed on Sui. USDC transfer and commerce artifact issuance (e.g., receipt/warranty objects) occur in the same state transition, giving strict cryptographic atomicity and deterministic reconciliation keyed by the intent hash.

2. **Phase 2: identiPay Solver Network (cross-rail fulfillment).** Once the protocol is stable on its "home rail," identiPay introduces a decentralized network of **Solvers** (liquidity providers and execution agents) that can *fulfill* a signed UIO even when the payer's funds live

elsewhere.**The merchant still settles on their preferred rail**, and the commerce artifacts are still minted atomically on Sui; only the *source of liquidity* changes.

3. **How solvers abstract rails.** If the payer holds value on another chain or in a bank account, the payer signs an intent that authorizes a specified payment action with clear limits (amount, expiry, recipient, and nonce). A Solver fronts on the merchant's preferred rail to satisfy the condition immediately, then settles back by claiming the payer-authorized funds on the external rail. From the merchant's perspective, nothing changes: one integration, one asset (USDC), one canonical settlement environment (Sui).

The Atomicity Spectrum & Intent Solvers. Strict cryptographic atomicity is only possible when both legs of the transaction exist on the same ledger (Native Sui Mode). When the protocol abstracts settlement across asynchronous rails (e.g., SEPA), strict atomicity degrades. To preserve the user experience, identiPay implements *Economic Atomicity.* When a user authorizes a fiat intent, a Solver immediately fronts the required USDC/Warranty Object on Sui. The Solver absorbs the latency of the external bank rail, capturing a marginal spread for providing instant execution.

5 Federated Routing & Resolution

Interoperability relies on the custom URI schema: `URI = did:identipay:<hostname>:<transaction-id>`. Because commercial proposals are too large for QR codes, the URI serves as a routing pointer. When a user scans a QR, the wallet performs a deterministic resolution via an HTTPS GET request to: `https://<hostname>/api/identipay/v1/intents/<transaction-id>`.

To prevent DNS spoofing, wallets cross-reference the resolved `<hostname>` against a decentralized **Trust Registry** maintained on the Sui blockchain before any intent is signed.

6 Pillar I: Stealth Identity & Privacy

6.1 Objective

Avoid using IBAN-style identifiers and raw public keys as transaction identifiers in consumer commerce. A naïve approach—mapping a human-readable name to a fixed blockchain address—creates a surveillance vector: anyone who learns the name can observe every transaction, every receipt, and every warranty on-chain. The identifier used for routing must be *unlinkable* to the addresses that actually receive value and commerce artifacts.

6.2 The Stealth Meta-Address Primitive

identiPay introduces a **stealth meta-address** system: a human-readable name (e.g., `@krum.idpay`) resolves to a pair of cryptographic public keys, *never* to a blockchain address. Every incoming payment or commerce artifact is delivered to a fresh, one-time *stealth address* that only the recipient can detect and spend from. No two transactions share an address. No on-chain address is ever publicly linked to the name.

6.3 Identity Ingestion and Meta-Address Generation

Credential ingestion. The wallet ingests a credential via NFC (national eID/passport), QR-mediated issuance, or secure retrieval of an EUDI-aligned Verifiable Credential. identiPay assumes an identity issuance stack consistent with the OpenID for Verifiable Credential Issuance/Presentation (OpenID4VC family).

Key generation. From a master seed, the wallet derives:

- an Ed25519 **spending keypair** $(k_{\text{spend}}, K_{\text{spend}})$ for signing transactions, and
- an X25519 **viewing keypair** $(k_{\text{view}}, K_{\text{view}})$ for detecting incoming payments.

The tuple $(K_{\text{spend}}, K_{\text{view}})$ constitutes the user's *meta-address*.

Identity commitment. The wallet computes a Poseidon hash commitment over the credential fields and a user-chosen salt:

$$C = \text{Poseidon}(\text{IssuerCertHash},\ H(\text{DocNumber}),\ H(\text{DOB}),\ \text{Salt}).$$

This commitment proves that a real government credential backs the registration, without revealing which one.

ZK proof of credential validity. Locally, the wallet generates a Groth16 proof attesting:

$$\pi_{\text{reg}} \leftarrow \text{Prove}(\text{CredentialValid} \wedge \text{Commitment} = C).$$

6.4 Name Registration

The user selects a human-readable name (e.g., `krum`). The wallet submits a registration transaction to the on-chain `MetaAddressRegistry` containing:

- the chosen name (3–20 characters, alphanumeric and hyphens),
- K_{spend} (Ed25519 public key, 32 bytes),
- K_{view} (X25519 public key, 32 bytes),
- the identity commitment C, and
- the Groth16 proof π_{reg}.

The on-chain registry enforces *one commitment per name*: a single government credential can register exactly one name, preventing Sybil attacks. The registry stores **public keys only**—no Sui address is recorded.

6.5 Stealth Address Protocol

Sending to a name. When a sender wishes to pay `@krum.idpay`, the sender's wallet performs the following off-chain:

1. Resolve the name from the on-chain registry: $(K_{\text{spend}}, K_{\text{view}})$.
2. Generate an ephemeral keypair $(r, R = r \cdot G)$.
3. Compute the ECDH shared secret: $\text{shared} = r \cdot K_{\text{view}}$.
4. Derive a scalar: $s = \text{SHA256}(\text{shared} \parallel$ `"identipay-stealth-v1"`$)$.

5. Compute the stealth public key: $K_{\text{stealth}} = K_{\text{spend}} + s \cdot G$.
6. Derive the stealth Sui address: $\text{addr} = \text{BLAKE2b-256}(\texttt{0x00} \parallel K_{\text{stealth}})$.
7. Transfer value to addr.
8. Emit a `StealthAnnouncement` event containing R, a one-byte *view tag* (first byte of shared), and the stealth address.

Detecting incoming payments. The recipient's wallet scans `StealthAnnouncement` events:

1. Fast-filter by view tag (reduces full ECDH computations by $\sim 256\times$).
2. Compute $\text{shared} = k_{\text{view}} \cdot R$ (ECDH with the viewing private key).
3. Derive s and $K_{\text{stealth}} = K_{\text{spend}} + s \cdot G$.
4. Check whether $\text{BLAKE2b-256}(\texttt{0x00} \parallel K_{\text{stealth}})$ matches the announced address.
5. If matched, derive the stealth private key: $k_{\text{stealth}} = k_{\text{spend}} + s$.

The recipient can now sign transactions from the stealth address.

Example: peer-to-peer payment. Bob wants to send 50 USDC to `@alice.idpay`. His wallet resolves the name from the on-chain registry and obtains Alice's meta-address—two public keys $(K_{\text{spend}}, K_{\text{view}})$. Crucially, this is all anyone can obtain from the registry; no Sui address is stored there.

Bob's wallet generates a fresh random scalar r (an ephemeral secret that exists only for this transaction) and computes the corresponding public point $R = r \cdot G$. Using r and Alice's viewing public key, the wallet performs an ECDH key exchange to produce a shared secret, then derives a one-time stealth address from that secret and Alice's spending public key. The wallet sends 50 USDC to this address and emits a `StealthAnnouncement` containing R, a view tag, and the stealth address.

Why can't an observer find Alice's other stealth addresses? The stealth address depends on r, which Bob chose at random and which is not published—only R is. Recovering the shared secret from R requires Alice's *private* viewing key k_{view}. A different sender, Carol, would pick a different random r', producing an entirely unrelated stealth address. An observer who sees both transactions sees two unconnected addresses with no on-chain link to `@alice.idpay` or to each other.

Alice's wallet, running in the background, scans each `StealthAnnouncement`. It checks the one-byte view tag first (a fast filter that eliminates ~ 255 out of 256 announcements without any expensive computation). For the remaining candidates, the wallet computes $\text{shared} = k_{\text{view}} \cdot R$. Because $k_{\text{view}} \cdot R = k_{\text{view}} \cdot r \cdot G = r \cdot K_{\text{view}}$, Alice recovers the same shared secret Bob used. She derives the stealth address, confirms it matches, and computes the stealth private key $k_{\text{stealth}} = k_{\text{spend}} + s$. She can now spend the 50 USDC whenever she chooses.

Example: merchant checkout. Alice visits a coffee shop and scans a QR code displayed on the merchant's POS terminal. Her wallet resolves the QR URI, retrieves the commerce proposal (items, price, warranty terms), and renders it in plain language: "Send 5 USDC to CoffeeShop. Receive: receipt."

Alice confirms. Her wallet self-derives a fresh stealth address for receiving the receipt—since she controls both keys, no announcement or external lookup is needed. She signs the intent hash and submits a PTB that atomically transfers 5 USDC to the merchant and mints a `ReceiptObject` to her stealth address. The receipt lands at a one-time address that cannot be linked to `@alice.idpay` by any on-chain observer.

Commerce artifact delivery. During merchant checkout, the buyer's wallet self-derives a fresh stealth address for receiving commerce artifacts (receipts, warranties). The stealth address is included in the signed intent as the artifact delivery destination. No announcement is needed—the buyer already knows the derivation parameters.

6.6 Spending and Coin Management

Value accumulates across many stealth addresses. Naïve merging—combining coins from multiple stealth addresses in a single transaction—creates on-chain linkage: the merged addresses are revealed to share an owner. Worse, transitive closure compounds this over time: if cluster $\{A, B, C\}$ and cluster $\{C, D, E\}$ share address C, all five are linked. Eventually, all stealth addresses collapse into one identifiable cluster.

identiPay avoids this by routing multi-address merges through a **shielded pool**:

- **Single-address spending:** when one stealth address holds sufficient balance, the wallet signs directly with that stealth private key. Change goes to a new self-generated stealth address. No privacy loss occurs.
- **Multi-address spending (pool-on-merge):** when no single address has enough, the wallet deposits USDC from each stealth address into the shielded pool contract individually, then withdraws the combined amount to a fresh stealth address via a ZK proof of note ownership. The pool breaks the on-chain link between the input addresses.

6.7 The Shielded Pool

The shielded pool is an on-chain contract that holds fungible tokens (USDC, SUI). Users deposit and withdraw via zero-knowledge proofs, breaking the link between deposit and withdrawal addresses.

Deposit. A user sends `Coin<USDC>` to the pool contract along with a note commitment $c = \text{Poseidon}(\text{amount}, \text{ownerKey}, \text{salt})$. The commitment is appended to an on-chain Merkle tree. The note is only spendable by the holder of ownerKey.

Withdraw. The user generates a Groth16 proof attesting:

$$\pi_{\text{pool}} \leftarrow \text{Prove}\Big(\exists \text{ note} \in \text{MerkleTree} : \ \text{note.amount} \geq W \ \wedge \ \text{nullifier} = \text{Poseidon}(c, \text{ownerKey})\Big)$$

where W is the withdrawal amount. The proof reveals the Merkle root, the nullifier (preventing double-spend), the recipient address, and the withdrawal amount. It does *not* reveal which note was spent or who owns it. The contract verifies the proof via `sui::groth16`, checks the nullifier against a spent-set, transfers W to the recipient, and (if change exists) inserts a new note commitment for the remainder.

What an observer sees. USDC flows into the pool from various addresses. USDC flows out to various addresses. The ZK proof guarantees no inflation and no double-spending, but the observer cannot determine which deposit funded which withdrawal.

Anonymity set. The pool's privacy strength grows with usage. With few users, timing and amount correlation can narrow the linkage probabilistically. Mitigations include batch processing of deposits and withdrawals on a fixed schedule (every N blocks) rather than immediately, and the natural growth of the anonymity set as identiPay adoption increases. Even with a small anonymity set, the pool provides strictly better privacy than direct merging, which produces 100% certain linkage.

6.8 Deposits and On-Ramp

To add funds, the wallet generates a fresh one-time deposit address (a self-derived stealth address using a counter-based ephemeral key). This address is displayed as a QR code or copyable string. Each invocation produces a different address, so the funding source (exchange, bridge, peer) cannot correlate successive deposits. The wallet recognizes deposits immediately since it holds the private key.

6.9 Gas Sponsorship

Spending from a stealth address requires native gas tokens. Sui's *sponsored transactions* allow a gas sponsor to pay gas on behalf of any address without learning the identity behind it. This is the recommended approach, as it introduces no linkability.

6.10 Viewing Key Delegation

The viewing key k_{view} can be shared selectively:

- **Auditors and tax authorities:** the viewing key grants read access to all incoming transactions without spending authority.
- **Per-context sub-keys:** future versions may support hierarchical derivation of scoped viewing keys for specific merchants or time ranges.

The spending key k_{spend} never leaves the device.

6.11 Eligibility Proofs (ZK Predicates)

For constrained transactions (e.g., age-gated purchases), the wallet generates a Groth16 proof attesting to predicates over credential attributes without revealing them:

$$\pi \leftarrow \text{Prove}(\text{Age} > 18 \wedge \text{CountryCode} \in \text{EU} \wedge \text{CredentialValid}).$$

The proof is bound to the intent hash, preventing replay. On-chain verification uses the `sui::groth16` native module. The verifier learns the truth of the predicate but not the underlying attributes, and critically, the proof does *not* reveal which registered name or meta-address the prover holds.

6.12 Security and Privacy Properties

1. **No address reuse:** every payment uses a fresh stealth address.
2. **Name $\neq$ address:** the on-chain registry maps names to public keys, never to Sui addresses. An observer who knows the name cannot find any associated on-chain address.

3. **Unlinkable transactions:** single-address spends reveal nothing; multi-address merges are routed through the shielded pool, breaking on-chain linkage via ZK proofs. Without the viewing key, stealth addresses belonging to the same user are computationally indistinguishable from those of different users.
4. **Minimized disclosure:** only predicate truth is revealed by ZK proofs; no PII is on-chain.
5. **Selective disclosure:** the viewing key enables auditing without spending authority.
6. **Anti-Sybil:** one government credential registers exactly one name (enforced by the identity commitment uniqueness constraint).
7. **Domain separation:** the same credential produces different commitments for different protocols, preventing cross-platform correlation.
8. **Replay resistance:** proofs bind to session parameters (intent hash, expiry, merchant DID) to prevent reuse outside the intended context.

7 Pillar II: Push-Only Intent Engine

7.1 Objective

Invert the default control model: merchants do not obtain a standing ability to pull funds. They can only publish *proposals*. The payer retains a unilateral technical stop mechanism by simply not producing a signed acceptance.

7.2 Proposal object

A merchant constructs a structured proposal (canonicalized JSON-LD or equivalent typed data), containing at minimum:

Field	Semantics
`sku`	item identifier(s), quantity, unit price
`amount`	exact payment amount and currency (e.g., USDC/EURC)
`merchant_did`	merchant identity (DID, certificate chain, or on-chain object)
`deliverable`	warranty/receipt/invoice metadata schema version
`expiry`	last valid acceptance time
`constraints`	policy (e.g., age-gate, region restriction)

7.3 Wallet parsing and human interface

The wallet must render the proposal into a *human-auditable* statement, e.g.:

> Send 50 USDC to Merchant X. Receive a receipt; age proof required. Expires in 10 minutes.

This translation is part of the security model: the UI must be driven by the typed proposal, not by ad hoc decoding of raw call data.

7.4 Semantic binding: intent to execution

The payer signs a domain-separated intent message, not opaque bytes:

$$\text{tx} \leftarrow \text{BCS}(\text{CanonicalProposal}, \text{Context}, \text{Nonce}), \qquad m \leftarrow \text{Intent} \parallel H(\text{tx}), \qquad \sigma \leftarrow \text{Sign}_{sk}(m).$$

On-chain execution is accepted if and only if the chain verifies
sigma over m and then executes the PTB contained in tx.

7.5 Revocation and "kill switch"

Because authorizations are expressed as *explicit acceptances* and optional scoped delegate capabilities (per-merchant/per-subscription), the payer can revoke a delegate key or disable a spending policy. This converts cancellation from a social process into a deterministic state transition.

8 Pillar III: Atomic Contextual Settlement on Sui

8.1 Objective

Couple value transfer and commerce artifacts as a single atomic outcome: it must be impossible to pay without receiving the warranty/receipt (or vice versa). The protocol should not rely on off-chain promises.

8.2 Sui PTBs as the settlement layer

Sui's object-centric model and programmable transaction blocks enable multi-step, multi-object transactions that either succeed entirely or fail entirely. identiPay uses PTBs to enforce conditional exchange among:

- a payment coin object (e.g., `Coin<USDC>`),
- a commerce artifact object (warranty/receipt/invoice), and
- optional compliance objects (tax logs, attestations).

8.3 Reference Move interface

The following snippet is illustrative (not a complete module):

```
public entry fun execute_commerce(
    payment: Coin<USDC>,
    merchant: address,
    buyer_stealth_addr: address, // one-time stealth address for artifact delivery
    zk_proof: vector<u8>,
    intent_sig: vector<u8>,
    intent_hash: vector<u8>,
    encrypted_payload: vector<u8>, // AES-256-GCM ciphertext (items + warranty)
    payload_nonce: vector<u8>, // 12-byte GCM nonce
    ephemeral_pubkey: vector<u8>, // E = e*G (for merchant decryption)
    has_warranty: bool,
    ctx: &mut TxContext,
) {
    // 1. Verify ZK proof of buyer eligibility (predicate truth only)
    verify_proof(zk_proof, intent_hash);

    // 2. Verify semantic binding: user signed the intent
    verify_intent_signature(intent_sig, intent_hash);

    // 3. Transfer payment to merchant
    transfer::public_transfer(payment, merchant);

    // 4. Mint receipt to buyer's stealth address
    // Payload is encrypted -- only buyer (k_stealth) and merchant
    // (k_merchant) can derive the decryption key
    let receipt = mint_receipt(
        merchant, intent_hash,
        encrypted_payload, payload_nonce, ephemeral_pubkey, ctx
    );
    transfer::transfer(receipt, buyer_stealth_addr);

    // 5. Mint warranty to buyer's stealth address (if applicable)
    if (has_warranty) {
        let warranty = mint_warranty(
            merchant, intent_hash, ephemeral_pubkey, ctx
        );
        transfer::transfer(warranty, buyer_stealth_addr);
    };

    // 6. Emit settlement event (indexed by intent hash, not by buyer identity)
    emit_settlement_event(intent_hash);
}
```

8.4 Determinism and failure modes

The transaction fails under any of the following:

- expired proposal or mismatched pricing fields,
- invalid or replayed intent signature,
- invalid ZK proof (or proof not bound to the intent hash),
- missing or malformed warranty/receipt object,
- policy constraints unsatisfied (e.g., age requirement).

This shifts the ambiguous, dispute-heavy domain of commerce toward explicit, checkable rules.

9 Data Model: Commerce Artifacts as Objects

9.1 Warranty and receipt objects

identiPay represents commerce artifacts as first-class objects with explicit schemas and evolvable versions. A minimal receipt/warranty object contains:

- **Merchant identity reference** (DID or on-chain object ID),

- **Payer reference** (stealth address—unlinkable to the payer's registered name),
- **Line items** and amounts,
- **Warranty terms** (duration, exclusions, service endpoints),
- **Integrity commitments** (hashes of off-chain documents if needed),
- **Transferability policy** (soulbound vs. transferable).

9.2 Artifact Encryption

Commerce artifacts (receipts, warranties) are delivered to stealth addresses, which prevents linking them to the buyer's name. However, if their contents—merchant name, line items, amounts, warranty terms—are stored as plaintext on-chain, any observer can read them. A receipt at an anonymous address that says "Laptop, 999 USDC, from ElectroShop" leaks the same information stealth addresses are designed to protect. identiPay therefore encrypts artifact payloads so that only the buyer and the merchant can decrypt them.

Key derivation. The buyer's wallet derives a deterministic ephemeral scalar from the stealth private key:

$$e = \mathrm{SHA256}(k_{\mathrm{stealth}} \parallel \texttt{"identipay-artifact-eph-v1"}), \qquad E = e \cdot G.$$

The wallet then performs ECDH with the merchant's public key K_{merchant} (obtained from the Trust Registry or the proposal):

$$\mathrm{shared} = e \cdot K_{\mathrm{merchant}}, \qquad K_{\mathrm{enc}} = \mathrm{SHA256}(\mathrm{shared} \parallel \texttt{"identipay-artifact-enc-v1"}).$$

The wallet encrypts the artifact payload (items, amounts, warranty terms) using AES-256-GCM with K_{enc} and a random 12-byte nonce. The on-chain artifact stores the ciphertext, the nonce, and the ephemeral public point E.

Decryption by the buyer. The buyer holds k_{stealth} and can recompute e, then K_{enc}, and decrypt. No additional state needs to be stored—the stealth private key is sufficient.

Decryption by the merchant. The merchant holds k_{merchant} and computes the same shared secret:

$$\mathrm{shared} = k_{\mathrm{merchant}} \cdot E = k_{\mathrm{merchant}} \cdot e \cdot G = e \cdot K_{\mathrm{merchant}}.$$

From this the merchant derives K_{enc} and decrypts the artifact. The merchant learns the receipt contents but does *not* learn the buyer's stealth private key, spending key, or registered name.

What remains in plaintext. The intent hash and the merchant's on-chain address are stored unencrypted, since they are required for on-chain settlement verification. The settlement contract validates the intent signature and ZK proof against the intent hash without needing to read the artifact contents. Everything specific to the commerce context—what was bought, for how much, under what warranty—is encrypted.

Selective disclosure for disputes. If a buyer needs to prove a warranty claim to a third party (e.g., a repair centre), the wallet can derive K_{enc} and share it alongside the artifact, granting read access to that specific artifact without exposing the stealth private key or any other transaction.

9.3 Ownership semantics

For consumer goods, warranties may be:

1. **Soulbound:** bound to a stealth address (and thus to the holder of the corresponding spending key) for non-transferable services.
2. **Transferable:** bound to the asset and transferable on resale.

The protocol treats this as policy encoded in object capabilities, not as a merchant-controlled database entry.

10 Use Cases

10.1 Subscription "kill switch"

Legacy. Cancellation is a human process; pull authority remains until a merchant updates its internal state.

identiPay. Subscriptions are modeled as scoped delegate capabilities or recurring proposals. Revocation of the delegate key (or removal of an allow-list policy) halts future executions deterministically; subsequent merchant proposals fail on-chain.

10.2 Self-enforcing warranty

Legacy. Receipts are external artifacts; eligibility for repair is discretionary and error-prone.

identiPay. At a service desk, the wallet proves ownership of the warranty object and (optionally) eligibility predicates. The repair authorization becomes a function of object ownership and terms, not of paper receipts.

10.3 Automated corporate compliance

Transactions can carry structured metadata (VAT category, merchant identifier, cost center) as commitments referenced by on-chain events. Indexers bridge these events into ERP systems with deterministic reconciliation keyed by intent hash rather than manually typed receipt data.

11 Integration Strategy

11.1 Principle

"Web2 in the front, Web3 in the back": adoption should not require merchants to rewrite core commerce stacks in Move.

11.2 Implementation sketch

- **Frontend plugin:** intercepts checkout to generate a proposal object and display a QR/NFC handoff to the wallet.
- **Backend listener:** monitors Sui for atomic settlement events keyed by intent hash.
- **Settlement:** receives USDC/EURC on-chain; optional automatic off-ramp occurs outside protocol scope via regulated partners.

11.3 Implementation Strategy: The Sui-First MVP

While the protocol is rail-agnostic, the Minimum Viable Product (MVP) is natively anchored on the Sui Blockchain utilizing the USDC Corridor. Sui's object-centric architecture and PTBs provide the only Web3 environment where "Atomic Contextual Settlement" can be executed natively without third-party solvers. A minimal `identipay::settlement` Move module accepts a `Coin<USDC>`, transfers it to the merchant, and atomically mints a `ReceiptObject` delivered to the buyer's one-time stealth address—unlinkable to their registered name.

12 Threat Model and Security Considerations

12.1 Adversaries

- **Malicious merchant** attempting to overcharge, replay intents, or deny deliverables.
- **Phisher/malware** attempting to induce blind signing or exfiltrate keys.
- **Observer** attempting to correlate identities across merchants.
- **Insider** attempting to alter receipts, warranties, or accounting metadata post-settlement.

12.2 Protocol mitigations

1. **No pull authority:** merchants cannot charge; only accepted intents execute.
2. **Typed intent signing:** signatures bind to canonicalized proposal hashes.
3. **Atomic swap:** delivery is enforced by PTB-level invariants.
4. **Stealth privacy:** names resolve to meta-addresses, not on-chain addresses; every transaction uses a fresh stealth address; predicates are proven without attribute disclosure.
5. **Replay protection:** expiry and nonce binding prevent reuse.

12.3 Residual risks

Wallet compromise remains a critical risk. Binding the wallet to an EUDI-aligned credential can strengthen onboarding, recovery, and compliance, but it does not prevent an attacker who gains control of the signing keys or device from authorizing transactions. identiPay reduces the attack surface by making prompts human-auditable, but endpoint security is not eliminated. Additionally, off-chain off-ramp processes introduce regulated counterparty risk and are intentionally out of protocol scope.

13 Conclusion

identiPay is a protocol-level redefinition of commerce settlement. By eliminating pull authority, binding execution to human-auditable intent, delivering value and commerce artifacts to unlinkable stealth addresses, and enforcing atomic exchange of value and context, the system aligns security, privacy, and reconciliation with deterministic on-chain invariants. Human-readable names provide usability; stealth meta-addresses provide privacy; ZK proofs provide selective disclosure. The target is not incremental optimization of payments, but a composable, privacy-preserving commerce layer suitable for sovereign digital identity and programmable money.