NERO 402: An AA-native settlement scheme for x402

Marcus Souza

2026-04-29

Abstract The x402 V2 protocol defines an HTTP-layer payment mechanism in which merchants gate resources behind a `402 Payment Required` response and clients settle the request via a facilitator service. The reference implementation defines a single settlement scheme, `exact`, which relies on EIP-3009 `transferWithAuthorization` and therefore requires that the paying account be an externally-owned account capable of producing EIP-712 typed signatures. ERC-4337 smart contract wallets cannot satisfy this requirement, leaving a class of accounts unable to participate in x402-mediated payments.

This document specifies `aa-native`, an additional x402 settlement scheme that accommodates ERC-4337 accounts. In place of an EIP-3009 authorization, the payment payload carries a signed `UserOperation` whose `callData` invokes a settlement contract; the facilitator validates that the call resolves to the merchant's `PaymentRequirements` and submits the operation via an ERC-4337 bundler. A paymaster MAY sponsor the gas, in which case the paying account does not require a balance of the chain's native asset. A reference implementation is provided as the NERO 402 stack, comprising a settlement contract deployed on NERO Chain mainnet and testnet, a facilitator service, a four-package TypeScript SDK, and a public playground. This document specifies the scheme, characterizes its security model, situates it relative to the existing `exact` scheme, and outlines the path to upstream standardization through `coinbase/x402#639`.

Conformance keywords. "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as defined in RFC 2119 and RFC 8174.

1. Background

1.1 The x402 V2 protocol

The x402 V2 wire format, as defined in `@x402/core@2.11.0`, specifies three HTTP headers and three JSON document types. The `PAYMENT-REQUIRED` header carries a base64url-encoded JSON list of `PaymentRequirements` entries the merchant accepts. The `PAYMENT-SIGNATURE` header carries a base64url-encoded `PaymentPayload` supplied by the client on retry. The `PAYMENT-RESPONSE` header carries a base64url-encoded `SettlementResponse` attached by the merchant to a successful response.

A `PaymentRequirements` entry specifies the price, asset, payee, network, scheme name, and timeout. A `PaymentPayload` consists of an envelope `{x402Version, accepted, payload, extensions}` in which `accepted` repeats the merchant's `PaymentRequirements` and `payload` is a scheme-specific document; the wire format treats `payload` as opaque.

The protocol flow proceeds as illustrated in Figure 1.

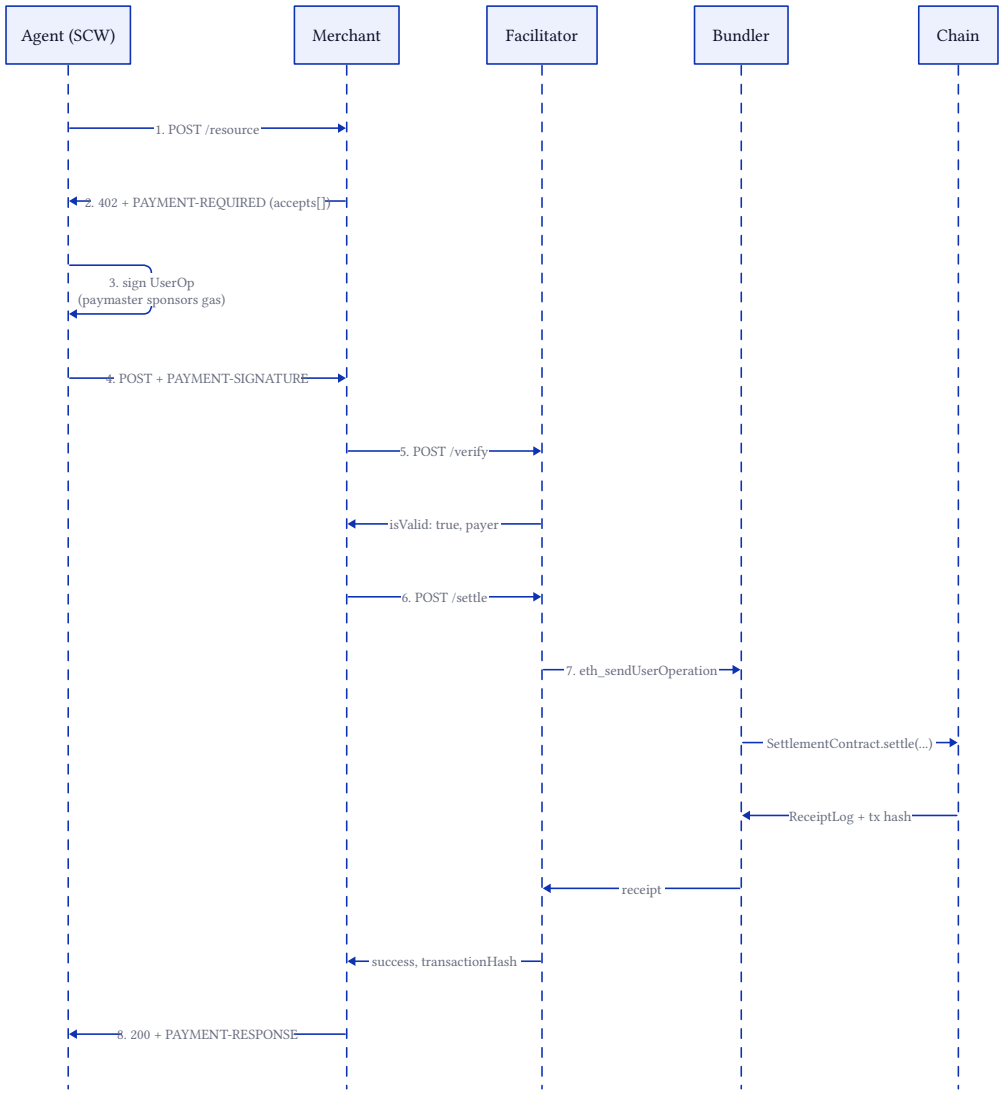

Figure 1: x402 V2 paid-call flow under the `aa-native` scheme.

Verification ensures the payload binds to the price the merchant offered. Settlement causes value to move on chain. The two operations are separated because verification is a read-only check

while settlement incurs on-chain cost; merchants commonly verify before invoking the protected handler and settle afterward.

1.2 Incompatibility between `exact` and ERC-4337 accounts

The `exact` scheme relies on EIP-3009 `transferWithAuthorization`. The paying account signs an EIP-712 typed authorization specifying the recipient, amount, and a nonce. The token contract recovers the signer address from a `(v, r, s)` tuple via `ecrecover` and verifies that this address holds the source balance.

`ecrecover` returns a 20-byte address derived from a 65-byte signature. This recovered address is, by construction, an EOA: the function cannot return a contract account. ERC-4337 accounts, by contrast, are CREATE2 deployments of contract code that perform their own signature validation, frequently via delegation to an owner EOA but optionally via session keys, multisig, social recovery, or other mechanisms encoded in the account contract.

Three approaches to reconciling these requirements have been considered:

1. The paying party operates both an EOA and an SCW, funds the EOA with the settlement asset, and signs EIP-3009 from the EOA. This approach satisfies the wire format but eliminates the operational benefits of using an SCW.
2. The token contract supports ERC-1271 (`isValidSignature`) in addition to `ecrecover`. The majority of widely deployed ERC-20 contracts, including Tether USDT on most chains, do not implement this fallback.
3. The settlement is moved off the token's authorization API and onto a settlement contract that the SCW invokes directly. The SCW signs a `UserOperation` whose `callData` targets the settlement contract; any standard ERC-20 supports this path without modification.

Approach (3) is the basis of the scheme defined in this document. The upstream issue tracking SCW participation in x402 is `coinbase/x402#639`.

1.3 Motivation

Two motivations apply to the design. The first is operational simplification for paying accounts. An account that holds only the settlement asset (typically a stablecoin) avoids the bookkeeping required to maintain a native-gas balance per chain. Native-gas balances must be replenished as they decay below bundler-imposed minima and reconciled across all chains the account interacts with. A paymaster that sponsors UserOp gas removes the native-gas balance entirely from the paying account's responsibilities.

The second is account portability. SCWs are addressed by their CREATE2 deployment recipe (`initCode`, factory, salt) rather than by funded EOA balances. A paying account can be reproduced from a private key and a constant; its on-chain state reconciles when the first UserOp lands. EOA-based settlement, by contrast, ties a paying account to a specific funded address per chain.

Whether either motivation justifies a new scheme rather than an extension to EIP-3009 is an open design question. The argument for a separate scheme is that EIP-3009 evolves only when a token issuer redeploys, which is uncommon; the argument against is that the wire format

gains complexity. This document advances the new-scheme position; the alternative remains a legitimate proposal upstream.

---

2. The `aa-native` scheme

2.1 Payload format

The payment payload uses the standard V2 envelope. The scheme-specific `payload` field carries two members, `userOp` and `settlementCallSpec`:

```
{
  "x402Version": 2,
  "accepted": {
    "scheme": "aa-native",
    "network": "eip155:1689",
    "amount": "1000",
    "asset": "0x926723bf5a2007012e3c609b53ae4a3b6c2dd1c8",
    "payTo": "0x1Ed81a78c7Ab41D17286E41473846Efe8A30c34f",
    "maxTimeoutSeconds": 60
  },
  "payload": {
    "userOp": {
      "sender": "0x...",
      "nonce": "0x...",
      "initCode": "0x",
      "callData": "0x...",
      "callGasLimit": "...",
      "verificationGasLimit": "...",
      "preVerificationGas": "...",
      "maxFeePerGas": "...",
      "maxPriorityFeePerGas": "...",
      "paymasterAndData": "0x...",
      "signature": "0x..."
    },
    "settlementCallSpec": {
      "merchant":   "0x1Ed81a78c7Ab41D17286E41473846Efe8A30c34f",
      "token":      "0x926723bf5a2007012e3c609b53ae4a3b6c2dd1c8",
      "amount":     "1000",
      "requestHash":"0x..."
    }
  },
  "extensions": null
}
```

The `accepted` block is the merchant's `PaymentRequirements` entry, replicated verbatim from the 402 response. The `payload.userOp` object is a complete ERC-4337 v0.6 `UserOperation` signed by the SCW's owner; its fields conform to the v0.6 layout in which `paymasterAndData` is a single `bytes` value rather than the split form introduced in v0.7. The `payload.settlementCallSpec`

object enumerates the merchant address, token address, atomic-units amount, and `requestHash` that the agent asserts the operation's `callData` will resolve to.

The `settlementCallSpec` member is informationally redundant: a verifier could derive each of its fields by ABI-decoding `userOp.callData` (§2.4). It is carried explicitly for two reasons. First, it permits the verifier to detect a payload-versus-call mismatch and return a precise error before incurring the cost of a bundler simulation. Second, it allows facilitators to attribute settled payments to merchants in audit logs without re-decoding `callData`.

2.2 Wire format integration

The `aa-native` envelope is structurally identical to `exact`; existing V2 clients and merchants route between schemes by inspecting `accepted.scheme`. Facilitators that support both schemes advertise this on `GET /supported`:

```
{
  "x402Version": 2,
  "schemes": [
    {"scheme":"exact",     "networks":["eip155:1689"], "assets":["0x...USDC",
"0x...USDT"]},
    {"scheme":"aa-native", "networks":["eip155:1689"], "assets":["0x...USDC",
"0x...USDT"]}
  ]
}
```

Clients select a scheme they are able to satisfy: an EOA-only client cannot use `aa-native`; an SCW-only client cannot use `exact`. Merchants that advertise both delegate the choice to the client.

2.3 `requestHash` derivation

The `requestHash` field binds a single payment authorization to a single (merchant, endpoint, time-window, nonce) tuple. It additionally serves as the on-chain replay key (§5.1).

```
requestHash = keccak256(abi.encode(
    merchantAddress,    // address
    chainId,            // uint256
    httpMethod,         // string, e.g. "POST"
    canonicalEndpoint,  // string, e.g. "402.nerochain.io/api/llm"
    timestampBucket,    // uint256, unix seconds floor-divided by 60
    clientNonce         // bytes16, random
))
```

The constituent fields prevent three classes of replay:

- **Cross-merchant replay.** A payload signed for merchant A produces a hash distinct from one a malicious party would require to settle the same authorization against a different merchant. The settlement contract enforces global uniqueness on `requestHash` (§4.1).
- **Stale-window replay.** A payload signed in a prior `timestampBucket` produces a different hash from a fresh one even when the `clientNonce` is held constant.

- **Same-window collision.** Two agents authorizing payments to the same merchant within the same bucket select independent random nonces, yielding distinct hashes with overwhelming probability.

The bucket size of 60 seconds reflects a tradeoff between the tightness of the replay window and tolerance for clock drift between agents and facilitators. Window equality is not enforced at verification time; the on-chain `isSettled[requestHash]` mapping (§4.1) is the authoritative replay check. The bucket exists to limit the residual utility of a leaked but unsubmitted authorization.

2.4 Verification

The facilitator endpoint `POST /verify { paymentPayload, paymentRequirements }` returns either `{isValid: true, payer}` or `{isValid: false, invalidReason, details?}`. A conformant verifier MUST execute the following checks before returning `isValid: true`:

1. **Envelope shape.** The `paymentPayload` MUST conform to `PaymentPayloadV2Schema`. Otherwise: reject as `invalid_envelope`.
2. **Scheme match.** `paymentPayload.accepted.scheme` MUST equal `"aa-native"` and MUST match the merchant's offered requirement. Otherwise: reject as `unsupported_scheme` or `requirements_mismatch`.
3. **Common fields.** The values of `accepted.network`, `accepted.amount`, `accepted.asset`, `accepted.payTo`, and `accepted.maxTimeoutSeconds` MUST equal the merchant's requirement, with address case normalized to EIP-55. Otherwise: reject as `network_mismatch`, `amount_mismatch`, `asset_mismatch`, or `payTo_mismatch`.
4. **Inner payload shape.** `payload.userOp` MUST conform to the ERC-4337 v0.6 `UserOperation` schema; `payload.settlementCallSpec` MUST contain all four required fields. Otherwise: reject as `invalid_inner_payload`.
5. **CallData decoding.** The verifier MUST ABI-decode `userOp.callData` and locate a `SettlementContract.settle(merchant, token, amount, requestHash)` invocation. Two outer call shapes are accepted: `SimpleAccount.execute(target, value, data)` and `SimpleAccount.executeBatch(targets[], datas[])`. For `executeBatch`, the verifier MUST scan the entries and select the first one whose `data` decodes to `settle(...)`. If neither shape parses or no `settle` invocation is located, reject as `calldata_decode_failed`.
6. **Spec equality.** The decoded `settle` arguments MUST equal `payload.settlementCallSpec` field-for-field. Otherwise: reject as `spec_mismatch`.
7. **Spec equals requirements.** `settlementCallSpec.merchant` MUST equal `accepted.payTo`; `settlementCallSpec.token` MUST equal `accepted.asset`; `settlementCallSpec.amount` MUST equal `accepted.amount`. Otherwise: reject as `spec_mismatch` (subcases: `merchant_mismatch`, `token_mismatch`, `amount_mismatch`).
8. **Replay registry.** The facilitator's off-chain replay registry MUST NOT have `requestHash` in the `settled` state. Otherwise: reject as `replay`.

A verifier MAY additionally pre-flight-simulate the `UserOperation` via `eth_estimateUserOperationGas`. The reference implementation does not do so, on the grounds that the bundler will surface the same revert during `/settle` and pre-flighting doubles bundler load.

When verification succeeds, the response carries a `payer` field equal to `userOp.sender` (the SCW's address). Merchants use this value for audit and observability; the verification result is the primary control-flow signal.

2.5 Settlement

The facilitator endpoint `POST /settle { paymentPayload, paymentRequirements }` re-runs verification and, on success, submits the `UserOperation` to a bundler. The settlement procedure MUST be idempotent on `requestHash`:

1. **Claim.** The settler MUST claim the `requestHash` in the off-chain replay registry by setting its state to `in_flight`. If the registry already records the hash as `settled` with an associated `txHash`, the settler MUST return a successful response with that `txHash` and MUST NOT resubmit. If the registry records it as `in_flight`, the settler MUST return `errorCode: "in_flight"` and MAY include a `Retry-After` header.
2. **Re-verify.** The settler MUST run the verification procedure of §2.4. On failure, the settler MUST release the `in_flight` claim and return the corresponding error code.
3. **Submit.** The settler MUST forward the `UserOperation` to the bundler via `eth_sendUserOperation` and capture the returned `userOpHash`.
4. **Wait.** The settler MUST poll `eth_getUserOperationReceipt` for a receipt within a configured timeout. The settlement contract emits `ReceiptLog(merchant, payer, token, amount, requestHash)` on success and reverts with `AlreadySettled(bytes32)` on a duplicate hash. A revert during execution causes the settler to return `errorCode: "user_op_failed"` with the surfaced revert reason.
5. **Complete.** On a successful receipt, the settler MUST transition the registry entry from `in_flight` to `settled` with the bundler-returned `transactionHash` and return the success envelope.

The success response has the form:

```
{
  "success": true,
  "x402Version": 2,
  "scheme": "aa-native",
  "network": "eip155:1689",
  "transactionHash": "0x...",
  "userOpHash": "0x...",
  "requestHash": "0x...",
  "payer": "0x...",
  "amount": "1000",
  "asset": "0x..."
}
```

The merchant attaches this body, base64url-encoded, as the `PAYMENT-RESPONSE` header on the eventual 200 response. Clients decode the header to obtain a structured proof of on-chain settlement, including the explorer-citable `transactionHash`.

2.6 Error codes

The verifier returns `invalidReason` strings; the settler returns `errorCode` strings. Both sets are stable identifiers intended for programmatic dispatch by merchants and SDKs.

The list below specifies each code, the subsystem(s) that may return it (V = `/verify`, S = `/settle`), and its meaning.

- `invalid_envelope` *(V)* — `paymentPayload` does not match the V2 envelope.
- `unsupported_scheme` *(V, S)* — facilitator does not handle this scheme.
- `requirements_mismatch` *(V, S)* — the `accepted` block does not match merchant requirements.
- `network_mismatch` *(V)* — `accepted.network` differs from the merchant's requirement.
- `amount_mismatch` *(V)* — `accepted.amount` differs from the merchant's requirement.
- `asset_mismatch` *(V)* — `accepted.asset` differs from the merchant's requirement.
- `payTo_mismatch` *(V)* — `accepted.payTo` differs from the merchant's requirement.
- `invalid_inner_payload` *(V)* — `payload.userOp` or `payload.settlementCallSpec` is malformed.
- `calldata_decode_failed` *(V)* — `userOp.callData` does not decode to `execute(...)` or to an `executeBatch(...)` containing a `settle(...)` invocation.
- `spec_mismatch` *(V)* — the decoded `settle` arguments differ from `payload.settlementCallSpec`.
- `replay` *(V, S)* — the `requestHash` has already been settled.
- `in_flight` *(S)* — another settlement for this `requestHash` is in progress.
- `bundler_error` *(S)* — the bundler rejected the UserOp (rate limit, simulation revert, or RPC error).
- `user_op_failed` *(S)* — the UserOp executed on chain but reverted.
- `receipt_timeout` *(S)* — the bundler did not return a receipt within the configured deadline.
- `internal_error` *(V, S)* — facilitator-side fault; the client SHOULD retry once.

Codes outside this list are reserved. Implementations introducing additional failure modes SHOULD allocate a new code rather than overload an existing one; collision with a future reserved code MAY occur otherwise.

---

3. Comparison to `exact`

The two schemes occupy distinct positions in the design space. The following table summarizes the principal axes of difference.

Dimension	`exact`	`aa-native`
Account type	EOA	ERC-4337 SCW
Authorization carrier	EIP-3009 typed signature on the token	UserOp signed by SCW owner; target = settlement contract
Required token feature	`transferWithAuthorization` (EIP-3009)	standard `IERC20.transfer` / `transferFrom`
Native-gas requirement	EOA must hold native gas	SCW does not require native gas if a paymaster sponsors
Settlement caller	facilitator-operated EOA submits the token call	bundler submits the UserOp; settlement contract calls the token
Replay protection	EIP-3009 nonce mapping at the token	`isSettled[requestHash]` mapping at the settlement contract
First-call cost	one ERC-20 transfer	one UserOp; if the SCW is undeployed, an additional ~170k gas for `initCode`
Token allowlist	implicit (any EIP-3009 token)	explicit (settlement contract owner-managed allowlist)
Wire format	V2 envelope; scheme-specific inner = `{signature, authorization}`	V2 envelope; scheme-specific inner = `{userOp, settlementCallSpec}`
Multi-chain	wherever the token's EIP-3009 is deployed	wherever an ERC-4337 stack is deployed

The `exact` scheme is operationally simpler when the paying account is an EOA and the asset supports EIP-3009. It bypasses the bundler entirely; the facilitator's submitter EOA pays gas in the chain's native asset. The end-to-end flow comprises two on-chain participants: the facilitator's submitter and the token contract.

The `aa-native` scheme is operationally simpler when the paying account is an SCW. The flow involves additional components (bundler, paymaster, settlement contract), but these components are part of the chain's existing ERC-4337 deployment and require no novel agent-side wallet logic.

The two schemes are not mutually exclusive at any layer of the protocol. A facilitator MAY implement both. A merchant MAY advertise both in `PAYMENT-REQUIRED.accepts[]`. A client selects whichever it can satisfy. The reference NERO 402 facilitator implements both for interoperability with unmodified upstream `exact` clients.

4. Reference implementation

The reference implementation, NERO 402, comprises four components: the on-chain settlement contract, the facilitator service, the SDK package suite, and the public playground.

4.1 SettlementContract.sol

The settlement contract is implemented in Solidity 0.8.28 against OpenZeppelin v5 contracts-upgradeable libraries. It uses the UUPS proxy pattern: an `ERC1967Proxy` is deployed in front of an upgradeable implementation that inherits `Initializable`, `OwnableUpgradeable`, `PausableUpgradeable`, `ReentrancyGuardUpgradeable`, and `UUPSUpgradeable`.

The contract maintains two state mappings and a reserved storage gap:

```
mapping(bytes32 requestHash => bool settled) public isSettled;
mapping(address token => bool allowed)        public isTokenAllowed;
uint256[48] private __gap;
```

The single state-changing function is:

```
function settle(address merchant, address token, uint256 amount, bytes32
requestHash)
    external
    payable
    nonReentrant
    whenNotPaused;
```

Validation is performed in the following order; any condition causes a revert:

```
if (msg.value != 0)              revert UnexpectedNativeValue();
if (merchant == address(0))      revert ZeroAddress();
if (token == address(0))         revert ZeroAddress();
if (amount == 0)                 revert ZeroAmount();
if (requestHash == bytes32(0))   revert ZeroRequestHash();
if (!isTokenAllowed[token])      revert TokenNotAllowed(token);
if (isSettled[requestHash])      revert AlreadySettled(requestHash);
```

Following validation, the function records `isSettled[requestHash] = true`, executes `IERC20(token).safeTransferFrom(msg.sender, merchant, amount)`, post-checks that the merchant's balance increased by exactly `amount`, and emits `ReceiptLog(merchant, msg.sender, token, amount, requestHash)`.

The `msg.sender` of `settle` is the paying SCW. The SCW MUST have approved the settlement contract for at least `amount` of `token` prior to the call. The reference SDK encodes this approval in the same UserOp via `executeBatch([approve, settle])`, ensuring atomicity between the approval and the transfer.

The owner-controlled token allowlist constrains the contract's exposure to malformed or hostile token implementations. The `_authorizeUpgrade` hook is owner-only and rejects the zero address. The `__gap` reserves 48 storage slots for future state additions, preserving the proxy's storage layout under upgrade.

Deployment addresses:

- Mainnet (`eip155:1689`): proxy `0x5eCfc64f2339992668f555918674B604F97B412D`, implementation `0xb7f16185619c476ce3fd3fd9e8b6186e340802f6`.
- Testnet (`eip155:689`): proxy `0x925dbba44570683ac8da99be08bc5ece8cf5a8c6`.

4.2 Facilitator service

The facilitator is implemented in TypeScript on Node 20+ using the Hono framework. It exposes four HTTP routes:

- `POST /verify` performs the verification procedure of §2.4 and returns `{isValid, payer}` or `{isValid: false, invalidReason}`.
- `POST /settle` performs the settlement procedure of §2.5 and returns a `SettleResult` envelope (success or error).
- `GET /supported` returns `{schemes: [...]}` advertising the schemes and assets the facilitator handles.
- `GET /health` returns `{status, network}`.

State storage in the reference implementation comprises a replay store keyed on `requestHash` with values recording `{status: in_flight | settled, txHash?}`. A token registry is constructed at boot from configuration; it maps lowercased token addresses to EIP-712 domain values consumed by the `exact` verifier. The `aa-native` verifier does not consume the registry directly but draws on the same allowlist for `/supported`.

External dependencies of the facilitator are the bundler RPC (for `eth_sendUserOperation` and `eth_getUserOperationReceipt`) and, indirectly, the paymaster RPC. The paymaster RPC is consumed by clients prior to payload submission; the facilitator forwards the resulting `paymasterAndData` field as part of the UserOp.

Authentication between merchants and the facilitator uses an HMAC-signed bearer token (`Authorization: Bearer <hmac>`). This authentication scheme is a facilitator-implementation detail and is not specified by x402.

Observability is provided by structured `pino` JSON logs, OpenTelemetry-compatible spans, and an off-chain indexer that ingests `ReceiptLog` events for a `/stats` endpoint.

4.3 SDK packages

Four packages are published under the `@nerochain/` scope:

- `@nerochain/x402-types`: Zod schemas and TypeScript types matching `@x402/core@2.11.0`. Exposes `PaymentPayloadSchema`, `PaymentRequirementsSchema`, `SettlementResultSchema`, and related definitions. Consumed by every other package in the suite.
- `@nerochain/x402-server`: middleware adapters for Express, Hono, Fastify, and Next.js App Router. Wraps a merchant's request handler, emits 402 on missing payment, forwards the payload to the facilitator on retry, and attaches `PAYMENT-RESPONSE` on a successful response.

- `@nerochain/x402-client`: an `x402Fetch({signer})` factory returning a `fetch`-compatible function that transparently handles 402 responses by signing and retrying. The signer is pluggable; the package contains no scheme-specific logic.
- `@nerochain/x402-aa`: an `aaNativeSigner({...})` factory returning a `PaymentSigner` for `x402Fetch`. Constructs the `UserOperation` via the NERO `userop` SDK, batches `[approve, settle]` calls, runs the paymaster middleware, signs, and returns the V2 envelope.

The agent-side surface required to issue a paid call comprises three import statements and one factory invocation:

```
import { ethers } from "ethers";
import { x402Fetch } from "@nerochain/x402-client";
import { aaNativeSigner } from "@nerochain/x402-aa";

const wallet = new ethers.Wallet(process.env.AGENT_KEY!);
const f = x402Fetch({
  signer: aaNativeSigner({
    signer: wallet,
    rpcUrl, bundlerUrl, paymasterUrl, paymasterApiKey,
    settlementContract: "0x5eCfc64f...",
  }),
});

const res = await f("https://402.nerochain.io/api/nero-knowledge", {
  method: "POST",
  body: JSON.stringify({ question: "How does the paymaster work?" }),
});
```

The merchant-side surface comprises a single middleware wrapper:

```
import { x402Next } from "@nerochain/x402-server/next";
export const POST = x402Next(
  { paymentRequirements, facilitator: { url } },
  async (req) => Response.json({ ... })
);
```

---

5. Security considerations

5.1 Replay protection

Replay protection operates at two layers. Only the on-chain layer is normative.

On-chain, the settlement contract maintains `isSettled[requestHash]`, a mapping initialized to `false` and set to `true` prior to each token transfer. A subsequent invocation of `settle` with the same `requestHash` reverts with `AlreadySettled(bytes32)`. This guarantee holds for the lifetime of the contract.

Off-chain, the facilitator maintains a replay registry that rejects already-settled `requestHash` values at `/verify` time, returning `replay` before incurring bundler simulation cost. The off-chain registry MAY lag the chain during confirmation windows or lead it during in-flight submissions;

consequently, it is not the source of truth. A facilitator operating with a wiped or out-of-sync registry cannot effect a double-spend, because the on-chain check rejects the duplicate. The off-chain registry is a latency optimization and not a safety mechanism.

5.2 Cross-merchant replay

The `requestHash` derivation (§2.3) includes `merchantAddress`. A payload signed for merchant A produces a hash that does not coincide with the hash a hostile party would require to settle the same authorization against a different merchant. Because `isSettled` is a global mapping in a single shared contract, even a hash collision (probability $\approx 2^{-256}$) admits at most one settlement.

The settlement contract's `settle` function takes `merchant` as an argument and forwards the token to that address. An attacker attempting to redirect funds by replaying the UserOp with a substituted `merchant` would alter the call's `keccak256` and consequently the `userOpHash`, invalidating the SCW owner's signature on the operation.

5.3 Paymaster abuse

A paymaster that sponsors UserOps is structurally exposed to abuse: an agent could submit operations that pass simulation, are sponsored, and revert on chain in a way that consumes paymaster gas without producing a useful settlement.

The reference paymaster's mitigations are:

- API-key-scoped sponsorship quotas enforced by the paymaster service.
- Bundler-side rate limits (on the order of 100 rps per API key under typical NERO infrastructure parameters).
- Facilitator-side rate limits on `/settle` (30 rps per merchant token, 10 rps per agent SCW).
- Off-chain replay registry rejection at `/verify`, prior to paymaster signature.

A paymaster operator MAY revoke a sponsoring API key out-of-band. The SDK does not assume an indefinite paymaster relationship. In deployments where merchants self-host facilitators, each merchant is responsible for the abuse posture of its associated paymaster.

5.4 Settlement finality

A settlement is considered finalized when the bundler returns a `UserOperation` receipt with `success: true`. The receipt includes the host transaction hash, after which the chain's standard confirmation rules apply.

A bundler-returned receipt is a trust point: the facilitator relies on the bundler's report. A maliciously colluding bundler could supply a false receipt. To mitigate this, merchants requiring strong settlement finality SHOULD independently verify by either of:

- Reading the transaction receipt directly via `eth_getTransactionReceipt` against an independent RPC.
- Watching for the `ReceiptLog` event on the settlement contract.

The reference Playground performs the latter check: its live ledger panel reads `ReceiptLog` events via `getLogs` polling, independent of the facilitator's reports.

5.5 Refunds

Refunds are out of scope for this specification.

The settlement contract has no refund function. A reverted UserOp does not move funds (the chain enforces atomicity). A successful UserOp with semantically incorrect arguments (for example, a typographically erroneous merchant address) moves funds the SCW intended to send. This behavior is equivalent to an erroneous `IERC20.transfer` and does not constitute a protocol-layer failure.

A future extension MAY introduce a time-locked dispute window with merchant-signed refunds, at the cost of additional contract and wire-format complexity. This specification takes the position that x402 is a payment protocol and not an escrow protocol; refund logic belongs at the application layer (for example, a merchant SDK that holds funds and forwards them only after delivering a successful response).

---

6. Future work and standardization

6.1 Multi-chain deployment

The `aa-native` scheme is chain-agnostic by design. Any chain supporting an ERC-4337 stack (EntryPoint, bundler, paymaster) can host a `SettlementContract` and serve x402 payments under this scheme. Each supported chain requires its own settlement contract deployment and its own merchant allowlist. The CAIP-2 network identifier in `accepted.network` makes the multi-chain dimension explicit at the wire level. A facilitator that handles multiple networks tracks one contract address per network.

6.2 Per-merchant settlement contracts

The reference implementation employs a single shared settlement contract for all merchants. This design minimizes deployment overhead and reduces audit surface, at the cost of representing merchants as arguments to `settle()` rather than as deployment-distinct recipients.

An alternative design adopts a factory pattern in which each merchant deploys an inexpensive cloned settlement contract. Such a design would enable per-merchant pause and upgrade controls, per-merchant token allowlists, and simpler accounting in cases where a merchant runs custom logic in the receive path. The tradeoff is increased deployment complexity and a larger contract surface to monitor.

6.3 Session keys and spending caps

In the present scheme, every UserOp is signed by the SCW's owner key. This requires the owner key to be online (or accessible to the agent process) at every payment.

ERC-4337 supports session keys via account-level validation logic. A session key delegated to an agent could carry constraints (for example, an upper bound on per-day spending against a specified `SettlementContract.settle` target on an allowlist of merchants) and could be revoked at any time.

Implementing this requires SCW-side validation logic outside the canonical `SimpleAccount`. Alternative account contracts (Kernel, Safe with modules, ZeroDev, and others) can be used today by substituting the factory and builder in the SDK; the `aa-native` wire format is unaffected. A future SDK package, `@nerochain/x402-aa-session`, could expose this as a configuration option without forking the package tree.

6.4 Standardization path

The `aa-native` scheme is currently specified by the present document alone. The path to incorporation into the x402 standard is:

1. **Open spec PR.** This specification, posted as a comment on `coinbase/x402#639` and as a follow-up PR proposing `specs/schemes/aa-native/` in the upstream repository.
2. **Community review.** Open discussion of the wire format, validation rules, and security model.
3. **Reference-implementation alignment.** If upstream review modifies the scheme, the reference implementation is updated to match. Upstream agreement is treated as the source of truth.
4. **Multi-implementation requirement.** Standardization of a new scheme typically requires a second independent implementation. An ERC-4337-native implementation on a different chain would satisfy this bar; absent such an implementation, the scheme remains a single-implementation extension.

The reference SDK is wire-compatible with the upstream `exact` scheme. The compatibility is verified by an interoperability test (`examples/express-paid-api/src/interop.test.ts`) that uses `@x402/core@2.11.0` to parse the reference middleware's 402 responses and submits Coinbase-format `PaymentPayload`s through the reference middleware.

Appendix A: Glossary

- **EOA** — Externally-owned account. An Ethereum account whose key is held off-chain.
- **SCW** — Smart contract wallet. An Ethereum account whose code is on-chain.
- **ERC-4337** — Account abstraction standard. Defines the `UserOperation` structure, the `EntryPoint` contract, and the bundler and paymaster roles.
- **UserOp** — `UserOperation`. The envelope an SCW signs to authorize an action.
- **Bundler** — Off-chain service that batches UserOps and submits them to the EntryPoint.
- **Paymaster** — Contract that pays gas for a UserOp on behalf of the SCW.
- **EIP-3009** — `transferWithAuthorization` standard. Defines an off-chain authorization signed by an EOA that a third party submits to move tokens.
- **EIP-712** — Typed structured data signing standard.
- **CAIP-2** — Chain-agnostic identifier format. NERO mainnet is `eip155:1689`; testnet is `eip155:689`.
- **Facilitator** — x402 service providing `/verify` and `/settle` on behalf of merchants.
- **`requestHash`** — Globally unique key for a settlement, derived from merchant, endpoint, time-window, and nonce.
- **Settlement contract** — On-chain contract that processes the token transfer and emits a structured event.

Appendix B: References

- x402 V2 launch: https://www.x402.org/writing/x402-v2-launch
- x402 HTTP transport: https://github.com/coinbase/x402/blob/main/specs/transports-v2/http.md
- x402 EVM `exact` scheme: https://github.com/coinbase/x402/blob/main/specs/schemes/exact/scheme_exact_evm.md
- AA support issue: https://github.com/coinbase/x402/issues/639
- ERC-4337: https://eips.ethereum.org/EIPS/eip-4337
- EIP-3009: https://eips.ethereum.org/EIPS/eip-3009
- NERO AA Platform documentation: https://docs.nerochain.io/en/developer-tools/aa-platform
- NERO UserOp SDK: https://docs.nerochain.io/en/developer-tools/user-op-sdk
- Reference implementation source: https://github.com/nerochain/nero-402
- RFC 2119, “Key words for use in RFCs to Indicate Requirement Levels”: https://www.rfc-editor.org/rfc/rfc2119
- RFC 8174, “Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words”: https://www.rfc-editor.org/rfc/rfc8174