CoRelayer

COOPERATIVE RELAYER INFRASTRUCTURE FOR AUTONOMOUS TRANSACTIONS

CoRelayer API
API overview

From the first request to a relayed transaction

REVISION	DATE
1.0	2 November 2026

ABOUT THIS DOCUMENT

CoRelayer API

From the first request to a relayed transaction

DOCUMENT	REVISION	DATE
API overview	1.0	2 November 2026
AUTHOR	**CONTACT**	**WEBSITE**
CoRelayer	support@co-relayer.com	co-relayer.com

ABSTRACT

The CoRelayer API from the first request to a relayed transaction: its surfaces, the concepts of a relay, authentication, payments, transaction states, the conventions shared by every route, the paths for AI agents and the SDKs. The reference at docs.co-relayer.com is authoritative.

NOTICE

The API reference at docs.co-relayer.com is authoritative for every route, field and error code. Prices and limits are set on chain and returned by the API. Built on MultiversX. CoRelayer is an independent project and is not affiliated with, endorsed by, or a product of MultiversX.

API OVERVIEW

Contents

01 What the API does 04

02 Core concepts 06

03 Authentication 10

04 A first relay 11

05 Plans and payments 14

06 Following what happens 16

07 Conventions 18

08 For AI agents 20

09 SDKs 22

10 Where to go next 23

11 Glossary and references 24

01

What the API does

The CoRelayer API is organised around REST. It takes and returns JSON, and reports every error as **problem details**, the format RFC 9457 defines. Through it, CoRelayer pays the network fee on your users' transactions: they sign once, in the wallet they already use, and you pay one plan in USDC.

The fee is paid by a **relayer**, a MultiversX wallet CoRelayer operates that adds its signature to the transaction. Four kinds of client call the API. A server relays the transactions its users sign, and one plan pays for all of them. A program that holds its own keys, such as a bot, a device or a script, sends from wallets named on its account. An AI agent finds CoRelayer, buys a plan and relays with no person involved. The dashboard at `app.co-relayer.com` reads an account's plan, usage and keys through the same API.

Surfaces

One process serves every surface (Table 1), and every address uses HTTPS. **Discovery files**, machine-readable files on co-relayer.com, tell software what CoRelayer is, what it costs and where it lives. Beside the REST API, three adapters stand in front of the same relay pipeline:

- **x402**, an open protocol that turns HTTP's 402 Payment Required into a payment exchange, lets an agent buy credits or a plan without an account.
- The **Model Context Protocol (MCP)** server lets an AI agent read CoRelayer's prices, network status, relayers and its quota as tools.
- The **compatibility routes**, three unversioned routes at the API's root shaped like a widely used open-source agent relayer interface, let such a kit switch by changing one base URL.

SURFACE	HOST	PATHS
REST API	`api.co-relayer.com`	`/v1/…`
HTTP 402 purchases	`api.co-relayer.com`	`/v1/x402/…`, `/.well-known/x402`
MCP server	`mcp.co-relayer.com`	`/mcp`, `/readonly`
Compatibility routes	`api.co-relayer.com`	`/health`, `/relayer/address/{userAddress}`, `/relay`
Discovery files	`co-relayer.com`	`/llms.txt`, `/pricing.json`, `/.well-known/corelayer.json`
OpenAPI document	`docs.co-relayer.com`	`/openapi.yaml`

Table 1 The surfaces of the API and where each is served.

CoRelayer documentation: API overview

Public reads

Reads that mirror the chain or a published document need no credential. These are the ones a client meets first:

- `GET /v1/network`: the **chain id**, the network identifier inside every signed MultiversX transaction, with the contract's address and the constants a transaction needs, from gas prices to the server's clock
- `GET /v1/pricing`: every plan and the price per transaction, as data (chapter 5)
- `GET /v1/relayers` and `GET /v1/relayers/{address}`: the relayer list as the API mirrors it, with each relayer's state
- `GET /v1/status`: the state of the service, component by component

Every CoRelayer relayer is listed on chain, and the API's list is a copy for convenience. Pin the contract address and the chain id in your own configuration: the chain id of MultiversX is `1`. Compare it with every transaction before you sign it, because an address alone does not identify a network. Where a discovery file disagrees with your pin, refuse it. The contract's address is published in `co-relayer.com/.well-known/corelayer.json`.

The reference and this overview

One OpenAPI 3.1 document describes the API. The reference, the types of the four SDKs and a plain-Markdown summary for agents are generated from it, and it is the authority for every route, field and error name. It is published under the Apache License 2.0, like the SDKs.

This overview explains how the parts fit together. Chapters 2 and 3 set out the concepts and the credentials. Chapter 4 walks one relay from start to end. Chapters 5 to 7 cover payments, following a transaction and the conventions every route shares. Chapter 8 is for agent builders, chapter 9 for SDK users, and the glossary at the end defines every term set in bold.

02

Core concepts

A relay rests on a few ideas: the account that pays, the unit it pays in, the relayer assigned before signing, the life of the transaction and the rules MultiversX sets.

2.1 Accounts, plans and Relay Units

A plan belongs to an account: the MultiversX address that bought it. Each plan covers a set number of transactions every 30 days, for as many users as you have, and is paid in USDC. The API names a plan by its `tierId`: 1 to 5 for Starter, Builder, Growth, Scale and Enterprise, and 11 to 13 for Agent Metered, Agent Pro and Agent Fleet. A period is given in milliseconds: `periodMs` is `2592000000`.

A payment becomes credits inside the contract. Credits buy plans and extra transactions, and they are not converted back to USDC. When a plan's transactions run out, sending pauses, so nothing is spent without your say. With pay-as-you-go on, you keep going at the extra-transaction price, from a budget set aside from your credits. `GET /v1/account/{erd}/quota` says where an account stands; `halted` is true while sending pauses.

The API's billing unit is the **Relay Unit (RU)**: an ordinary transaction is one, and an unusually heavy one counts as a few. One is counted for every 0.001 EGLD of network fee a transaction could reach at most, rounded up. The rule, its rounding and its test vectors are published, and a fingerprint of that document is stored in the contract, with 48 hours' notice before it can change.

Every purchase and renewal carries a price ceiling you sign, and a renewal above it is skipped, not charged. A purchase carries the quote's `maxPrice` as its `max_price`; automatic renewal needs its own, `max_renew_price`, where zero never means unlimited. A **quote** is the API's statement of what something costs, and a purchase quote is valid for 120 seconds.

2.2 Who pays

Your plan sets how many transactions, not how many users. An account pays for transactions that other addresses sign in one of two ways:

- A **sponsor key** is a secret key on the customer's server that makes the plan pay for transactions any user signs, for the listed contracts. The server sends it as `X-Api-Key`.
- **Named wallets** are addresses recorded on chain against an account, so the account's plan pays for their transactions. The API calls them authorised senders, listed by `GET /v1/account/{erd}/senders`.

The payer is decided in a fixed order: the sponsor key, if one is sent; the `account` the request names, if it listed the sender; the sender's own account; then the accounts that listed the sender, oldest first. The answer to a relay names the account billed in `account`, and how it was found in `billing.authMode`.

An account that can pay is billed in Relay Units for every relay, calls to the CoRelayer contract included. Only when no account can be billed (no plan, a plan that has lapsed or paused, or no account at all) is a short list of calls to the CoRelayer contract relayed at no charge, so that buying and managing a plan never needs one already. The pricing document publishes that list in `howToBuy.freeOperations`.

2.3 The assignment

The relayer is assigned just before your user signs: a healthy, funded relayer in the sender's shard, where MultiversX requires it to be. `POST /v1/relay/assign` returns that **assignment**: the relayer's address, which goes into the transaction before it is signed, and a lease, a short-lived token that goes back with it. An assignment is valid for 60 seconds and reserves nothing, so asking twice does no harm.

A lease that runs out while your user signs is renewed for the same relayer with `renewFor`, and the same signed bytes are sent again. Only when the assigned relayer is lost before it co-signs is a new signature asked for. It uses the same **nonce**, the sender's transaction counter, whose every value the network accepts once, so the old and the new transaction can never both run.

Times are Unix milliseconds, in members whose names end in `Ms`; correct your clock against `serverTimeMs`.

2.4 The life of a relay

An **intent** is one user-signed transaction CoRelayer was asked to relay, keyed by its sender and nonce; its id is `<sender>:<nonce>`. Delivery turns on one boundary: the moment the relayer's signature exists. Before it, nothing can run, and an error always means nothing was sent. After it, the transaction stays valid on the network until its slot is used, so the honest answer is a state (Table 2), never a failure.

STATE	MEANING
`ASSIGNED`	The client holds a lease. Nothing exists on CoRelayer's side.
`RECEIVED`	The API holds the bytes and is checking them.
`REJECTED`	Final. No signature was produced and nothing was sent.
`ACCEPTED`	The checks passed and the Relay Units are reserved.
`COSIGNED`	Past the boundary: the relayer's signature exists.
`BROADCAST`	An access point to the network has acknowledged the transaction.
`EXECUTED_OK`	In a block, and succeeded. It counts.
`EXECUTED_FAIL`	In a block, and failed. It counts, because the network charges its fee.
`EXPIRED_LOCALLY_STILL_VALID`	Sending has stopped. It stays valid until its nonce is used.
`DEAD`	Another transaction used the nonce. This one can never run; nothing counts.

Table 2 The states of an intent, in the API's names.

CoRelayer documentation: Intent lifecycle

A transaction that runs and fails counts: the network charges its fee, and we pay it. One that never runs doesn't count. `final` turns true once the block the transaction executed in is final in that block's shard, and that is when it is billed.

> **The client rule**
>
> On a timeout, send the identical bytes again or read the relay by its sender and nonce. Never rebuild and never re-sign. Re-sending identical bytes is the same transaction, so it is safe.

2.5 MultiversX rules the API applies

MultiversX sets the gas rules of every transaction, and CoRelayer adds bounds of its own before it takes on a transaction's fee (Table 3). `GET /v1/network` returns the values in force as `minGasPrice`, `maxGasPrice`, `minGasLimit`, `gasPerDataByte`, `extraGasRelayed` and `maxGasPerTransaction`; the pricing document publishes CoRelayer's bounds, with each plan's heaviest transaction in `maxRuPerTx`.

RULE	VALUE	SET BY
Lowest gas price	1,000,000,000	MultiversX
Highest gas price accepted	twice the lowest	CoRelayer
Smallest gas limit	50,000	MultiversX
Gas for each byte of data	1,500	MultiversX
Extra gas for a relayed transaction	50,000	MultiversX
Most gas in one transaction	600,000,000	MultiversX
Most data in one transaction	4,096 bytes	CoRelayer
Heaviest transaction	25 Relay Units; 15 on Starter and Agent Metered	CoRelayer
Guardians	a relayer is never guarded, nor a guarded sender's guardian	MultiversX

Table 3 The bounds every relayed transaction meets.

MultiversX documentation; CoRelayer documentation: Relay Units

Gas prices are in atto-EGLD per unit of gas. A transaction outside a bound is refused before anything is signed for it, with an error that names the bound, such as `GAS_PRICE_OUT_OF_RANGE` or `DATA_TOO_LARGE`. Guarded senders are served: add `extraGasGuarded` to the gas limit as well.

03

Authentication

There is no password and no sign-up. Most credentials are signatures you already make, and Table 4 shows which route reads which.

CREDENTIAL	SENT AS	READ BY
None	—	Public reads, quotes, validation, prepare
The transaction	the request body	`POST /v1/relay`
A presence proof	`proof` in the body, or native auth	`POST /v1/relay/assign`
Native auth	`Authorization` header	Private reads and account settings
A sponsor key	`X-Api-Key` header	Both relay routes; reads with the `read` scope
An x402 payment	`PAYMENT-SIGNATURE` header	`POST /v1/x402/topup`, `/purchase`
A stream ticket	`ticket` query parameter	`GET /v1/stream`, from a browser

Table 4 The credentials of the API and the routes that read them.
CoRelayer documentation: Authentication

A relay is authorised by the signed transaction itself: its bytes carry the sender's signature, name a CoRelayer relayer and use a nonce the network will accept. The route needs no token, and the nonce stops a replay.

Native auth is MultiversX's wallet-signature sign-in. The dashboard and private reads use it as a bearer token, which reads only the account whose address it carries. A token never travels in a URL (chapter 6). A dashboard sign-in lasts up to 24 hours. A program builds its own token from the origin and the recent block that `GET /v1/network` returns in `nativeAuth`.

A sponsor key is a secret shown once, and no key can create or revoke another; chapter 5 gives its policy. Keep it on your server, never in a web page; a request that carries a key and a browser's `Origin` header is refused. A key has the form `crk_live_…`.

An assignment needs proof that the sender is present, and the sender's own key always gives it: a signed, timestamped message that names the chain and the sender, or the sender's native-auth token. A program signs its bytes (`format: raw`); a browser wallet signs it as a MultiversX signed message that also names its audience, your web origin or `key:<keyId>` (`format: message`). A sponsor key adds the payer and never proves presence: `proof.kind` is `sponsor` exactly when `X-Api-Key` is sent, and `key` otherwise. The proof stops anyone who merely knows an address from reviving a transaction that was never relayed; the reference gives its bytes.

04

A first relay

A relay is four steps: ask for a relayer, sign once, send, follow. This chapter walks them as a server that pays for its users with a sponsor key, as Figure 1 shows; a program that sends from its own wallets takes the same steps without the key.



Figure 1 One relay paid by a sponsor key, from the assignment to the API's answer.

CoRelayer documentation: Pay for your users

1. Ask for a relayer

Ask just before your user signs. The request names the `sender` and carries the sender's presence proof, marked `sponsor`, and your server adds the sponsor key in the `X-Api-Key` header. Before your user signs anything, the API checks the key, its plan and its daily limits, and for a browser wallet's proof that the audience is one the key may serve.

The answer is the assignment. Table 5 lists the members your server uses.

MEMBER	USE
`relayer`	Put it in the transaction's `relayer` field before signing.
`lease`	Send it back with the signed transaction.
`leaseExpiresAtMs`	The end of the lease, 60 seconds after it was issued.
`chainId`	The transaction's `chainID`, after a check against the client's pin.
`minGasPrice`, `maxGasPrice`	The band the gas price must fall in.
`extraGasRelayed`, `extraGasGuarded`	Add them to the gas limit; the second only for a guarded sender.
`expectedNonce`	The nonce CoRelayer expects next from this sender.
`pinNonce`	On a re-sign, cancel or free lease: the one nonce it may sign.
`registryVersion`	The cache key for the client's check of the relayer on chain.
`sponsor`	The account that will pay and the key's id.

Table 5 The members of an assignment and what to do with each.

CoRelayer documentation: Shards and routing

2. Build and sign once

Build the transaction with the relayer in it: `relayer` set to the assigned address, the gas limit raised by `extraGasRelayed`, a gas price inside the band and the assignment's `chainId`.

Before your user signs, check the relayer against the on-chain list. The SDKs do it when you pass `verifyRelayer`; without an SDK, call the contract's `getRelayerState` view through a MultiversX node you choose, and sign only for an active relayer. Your user signs once, in the wallet they already use. The relayer's address is part of what they sign, so it cannot be swapped afterwards.

3. Relay with the sponsor key

Your server sends the signed transaction to `POST /v1/relay`, with your sponsor key in the `X-Api-Key` header. The body holds the whole signed transaction in `tx` and the assignment's `lease`. A lease made for a sponsor key is relayed only with that key. Add an **idempotency key**, one per user action, so that a

retry after a timeout cannot relay the same action twice. With the TypeScript SDK, the server's part is this region of a tested example:

```typescript
import { createHash } from 'node:crypto';
import { CoRelayerClient, type RelayRequest } from '@corelayer/sdk';

// On your server. The sponsor key never reaches a browser.
const apiKey = process.env.CORELAYER_API_KEY;
if (!apiKey) throw new Error('Set CORELAYER_API_KEY');
const corelayer = new CoRelayerClient({
  baseUrl: 'https://api.co-relayer.com',
  apiKey,
});
// Your user signed the transaction. Your plan pays the network fee.
export async function payForUser(req: RelayRequest, actionId: string) {
  const key = createHash('sha256').update(actionId).digest('hex');
  const { data } = await corelayer.relay(req, { intentKey: key });
  return data.txHash;
}
```

The API answers once the relayer's signature exists: `200` when an access point to the network has acknowledged the transaction, or `202` when it is co-signed and unacknowledged, while sending goes on. An error from this route always means nothing was sent.

The answer carries the intent's id in `intentId`, the `txHash`, the `state` and the Relay Units the transaction counts as in `ru`. It names the account billed in `account`. `billing.authMode` says how that account was found, `api_key` for a sponsor key, and `billing.ruReserved` gives the Relay Units the relay holds against the account's allowance until it is billed. `duplicate` marks a repeat of identical bytes, and `statusUrl` and `eventsUrl` say where to follow the relay.

4. Follow it to the end

`POST /v1/relay` never waits for a block. `GET /v1/relay/{id}/events` sends the whole intent each time it changes and closes once the intent is final, and `GET /v1/intents/{sender}/{nonce}` answers at any moment. Chapter 6 lists every way to follow a relay.

Before sending

Two routes check a transaction without sending it: `POST /v1/quote` prices it, and `POST /v1/validate` runs every check a relay would and sends nothing. To start small, Agent Metered has no monthly fee: you pay per transaction from the USDC you deposit.

The complete program, in TypeScript, Go, Rust and Python, is the recipe "Pay for your users", where every sample comes from a file that a test suite runs → docs.co-relayer.com/sdk/recipes/sponsor-users.

05

Plans and payments

Plans are priced from a fixed USDC price per transaction, set on chain. The API publishes that price and every plan as data, and builds every purchase as a transaction you sign once.

5.1 Buying is gasless

Buying a plan, renewing it and changing its settings are relayed by CoRelayer itself, so a wallet that holds only USDC can buy and manage a plan. Four `POST` routes prepare those transactions (Table 6). Each returns the unsigned transaction and a one-line summary, and says before your user signs how the call is paid. When the account can pay, `free` is false, `ru` gives the Relay Units it is billed, and its relayer comes from `POST /v1/relay/assign`; when no account can be billed, `free` is true and the answer carries the assignment to send it with. You sign it once and relay it like any other.

ROUTE	BUILDS	CONTRACT CALL
`/v1/subscribe/prepare`	A plan, with its quote and price ceiling	`depositAndSubscribe` or `subscribe`
`/v1/deposit/prepare`	Credits, from USDC	`deposit` or `depositFor`
`/v1/flags/prepare`	One setting change, or a renewal now	`setAutoRenew`, `setPayg`, `releaseEscrow`, `renew`
`/v1/senders/prepare`	Adding or removing named wallets	`addSenders`, `removeSenders`

Table 6 The prepare routes and the contract calls they build.
CoRelayer API reference: purchase

Adding a wallet you name beyond the ones the plan includes is the one paid action: a small one-time fee per wallet. The answer's `quote` says how many wallets the plan includes, how many new ones are charged and the whole fee, which the call signs as its ceiling. Removing one has no fee. Every payment is made in USDC, the token `USDC-c76f1f`. An agent can also buy over HTTP 402, with no account (chapter 8).

5.2 Prices and quotes

`GET /v1/pricing` returns the pricing document, and `co-relayer.com/pricing.json` serves the same document as a file. It carries every plan in `tiers` and the price per transaction in `tariff`, with any pending change and when it takes effect. It also holds the counting rule, the admission bounds and each

plan's speed limits. Its amounts are decimal strings in micro-USDC: parse them as integers, never as floating-point numbers. `GET /v1/pricing/tariff-history` lists every price per transaction ever in force.

Price rises are announced 48 hours ahead, and a period you've paid keeps its price. A rise shows in the pricing document, in your notices and in the dashboard the moment it is scheduled. A single payment has a ceiling the contract sets from the exchange's reserves (`deposit.maxMicroUsdc`), and a larger purchase is paid in several parts. The smallest payment is `deposit.minMicroUsdc`.

`POST /v1/quote` returns the Relay Units a transaction counts as and the most network fee it could reach. `POST /v1/validate` runs every check a relay would, with no lease or signature, and returns every problem it finds.

5.3 Quota, usage and proofs

What an account bought is public, because it mirrors the chain; what it used is private, because it is CoRelayer's record. Table 7 lists the `GET` routes that read them.

Settlement is the periodic contract call that debits pay-as-you-go budgets and commits the usage root of its window: a Merkle root over all its usage, written into the contract. A **usage proof** is a Merkle proof that a billed transaction is included in one of those roots, so what you were billed can be checked against the chain.

ROUTE	RETURNS	ACCESS
`/v1/account/{erd}/quota`	The service state; what the period has used, holds for relays in flight and has left; the pay-as-you-go budget and the next renewal	Public
`/v1/account/{erd}`	The account as the contract holds it: credits, plan and settings	Public
`/v1/usage`	Every transaction of the account, billed or not; `format=csv` exports it	Private
`/v1/usage/summary`	The totals of a period	Private
`/v1/usage/{txHash}`	One transaction's billing and settlement	Private
`/v1/usage/{txHash}/proof`	The usage proof of one billed transaction	Private

Table 7 The routes that read an account's plan and usage.

CoRelayer API reference: account, dashboard

5.4 Sponsor keys

Sponsor keys are created, changed and revoked with your wallet, through native auth: `GET` and `POST /v1/account/{erd}/keys`, `PATCH` and `DELETE /v1/account/{erd}/keys/{keyId}`. A key's policy lists the contracts it may pay for in `receiverAllowList`, which the `relay` scope requires, and can add `functionAllowList`, `maxRuPerSenderPerDay`, `maxRuPerDay` and `ipAllowList`. A leaked sponsor key is bounded by its required contract list, its daily limits and revocation. Sponsoring is on Builder, Growth, Scale, Enterprise, Agent Pro and Agent Fleet.

06

Following what happens

`POST /v1/relay` answers once the relayer has signed, and the block comes later. Follow one relay by its intent, or everything an account does on one connection.

One relay

Four `GET` routes read one relay, and each returns the same intent (Table 8).

ROUTE	GIVES
`/v1/relay/{id}/events`	Server-sent events, the whole intent at each change, until it is final
`/v1/relay/{id}`	The intent, by intent id or by transaction hash: the `statusUrl` of a relay's answer
`/v1/intents/{sender}/{nonce}`	The intent, authoritative at any moment: the answer to ask for after a timeout
`/v1/tx/{hash}`	The intent behind a transaction hash

Table 8 The routes that follow one relay.

CoRelayer documentation: Watch an intent

The event stream follows the intent whichever server relayed it, and closes once the intent is final, or after 120 seconds. Read nothing else while it delivers. When it ends, breaks or stays silent for 30 seconds, read `GET /v1/relay/{id}` once and, unless that read is final, open the stream again with `Last-Event-ID`. A final state comes from chain facts keyed by sender and nonce, never from a timeout: a timeout means unknown.

The account stream

`GET /v1/stream` sends everything one account does as server-sent events, under topics such as `intent`, `quota`, `account`, `notice` and `pricing`; its `intent` events cover every relay of the account, whichever server relayed it. A streaming HTTP client sends its native-auth token or a `read` key in a header. A browser's `EventSource` cannot send headers, so it asks `POST /v1/stream/tickets` for a single-use ticket and opens the stream with `?ticket=`.

`Last-Event-ID` resumes a stream; when the events after it are no longer held, the first event is `reset`, and the client reloads what it shows. Events are hints to read again: the routes stay the record.

Notices

Plan and price notices arrive in a feed your server can poll and on the account stream. `GET /v1/account/{erd}/notices` returns them from a `sinceSeq` you keep, and `POST /v1/account/{erd}/notices/ack` marks them read. Their kinds include `tariff.scheduled`, sent when a price rise is announced, `cap.threshold`, `renewal.skipped` and `service.halted`.

A webhook delivers the same notices to an HTTPS endpoint of yours, each signed with a secret that is shown once; `GET /v1/account/{erd}/webhooks/{webhookId}/deliveries` lists every attempt of the last 30 days. Notices can also go by e-mail to one address per account, which is verified by a link before anything else is sent to it. `PUT /v1/account/{erd}/notifications` sets the address and switches each of the four e-mail kinds: a renewal coming up, a plan that has lapsed, a plan's Relay Units used up, and the service's status notices.

07

Conventions

These rules hold for every route, so the reference does not repeat them on each page.

7.1 Requests and responses

Requests and responses are JSON in UTF-8, with member names in `camelCase`, and a write route refuses a member it does not know. Amounts of EGLD are decimal strings in atto-EGLD, in members ending in `Atto`. On the relay, account and usage routes, micro-USDC, Relay Units and gas are integers; in the pricing document, the quotes and the x402 challenges, money is a decimal string.

Every response carries two headers: `CoRelayer-Request-Id` names the request, and `CoRelayer-Version` names the API build. A problem's `instance` repeats the request id, the one to quote to support. A client may send `CoRelayer-Client: <name>/<version>`, which is logged and never used to refuse a request.

7.2 Errors

Every error carries a stable code and says whether to retry. It is a problem details document, `application/problem+json`, whose `type` is the address of the code's page in the catalogue at docs.co-relayer.com/errors.

Branch on `code`, never on `title` or `detail`, and only sign again when `resign` says so (Table 9). When a 429 or a 503 carries `Retry-After` or `details.retryAfterMs`, wait that long; otherwise back off with jitter, and send the same request again. A `503 UPSTREAM_UNAVAILABLE` for a relay, with a `details.reason` such as `HOST_BUSY` or `SLOT_CLAIM`, took nothing: nothing was reserved or co-signed, and the identical bytes may be sent again after `Retry-After`; the catalogue lists every such reason.

MEMBER	USE IT FOR
`code`	The branch: a stable machine name
`status`	The fallback for a code the client does not know
`retryable`	Whether the same request can succeed if sent again
`resign`	On the relay path: `NONE`, `SAME_BYTES` or `NEW_SIGNATURE_SAME_NONCE`
`hint`	One sentence that names the next action
`details`, `intent`	The specifics of the code, and the intent it concerns

Table 9 The members of a problem details document to branch on.

CoRelayer documentation: Errors and retries

7.3 Idempotency and pagination

Sending the same signed bytes twice is safe: the API knows them by sender, nonce and a hash of the bytes, and the second request gets the stored response, marked `duplicate: true`. An idempotency key adds one key per user action, sent as the `Idempotency-Key` header or as `intentKey`, so that a client cannot sign a fresh transaction for the same action by mistake.

The same key with other bytes or another nonce, while the first intent lives, gives `409 INTENT_ALREADY_SUBMITTED` with the first intent attached. Other bytes at a nonce that a live relay holds are refused with `409 NONCE_IN_FLIGHT`, so one nonce never carries two relays at once. Creating a sponsor key or a webhook requires an idempotency key.

Lists are paged with a **cursor**: send `limit` and `cursor`, and read `nextCursor` and `hasMore`. A cursor is opaque, does not expire and belongs to the filters it was made with; there are no offsets and no total counts. To keep up with a list, read on from where you stopped: `sinceSeq` for notices, `fromMs` for usage.

7.4 Limits and versions

Each plan has speed limits: how many Relay Units an account may send in a given time, and how much gas in each shard. They belong to the account, never to one sender, and the pricing document publishes them. Rate-limited routes send `RateLimit-Limit`, `RateLimit-Remaining` and `RateLimit-Reset`; when a 429 carries `Retry-After`, wait that long before you retry. `QUOTA_EXHAUSTED` carries none: waiting does not help, buying does.

The major version is in the path, `/v1`. A breaking change creates a new major version, and the old one keeps running alongside it. Inside `/v1`, changes only add routes, optional members and values of open enums such as `ErrorCode`, so ignore what you do not know and handle an unknown error code by its status and `retryable`. A route due for removal first carries `Deprecation` and `Sunset` headers, and the changelog records every change.

08

For AI agents

Give your agent USDC and a MultiversX key. It finds CoRelayer, picks a plan, pays and starts sending. No account and no dashboard. Its plan covers the network fee.

Four steps

Paths and fields come from the OpenAPI document. Your agent can read all of it first.

Discover 01

Three files on co-relayer.com state the price, the chain and the contract. No key is required. The chain id is pinned before signing.

Price 02

Plans and prices are returned as data. A quote gives the exact amount and the ceiling to sign.

Pay 03

Inside an HTTP 402 exchange, or by signing the contract call yourself. Same purchase, same price, either way. No account to create.

Relay 04

Request a relayer, name it in the transaction, sign once and submit. A platform's sponsor key (`X-Api-Key`) pays for all of its agents.

Discovery and pinning

Three files on co-relayer.com tell an agent what CoRelayer costs, which chain it is on and where the contract lives: `/llms.txt`, `/pricing.json` and `/.well-known/corelayer.json`. The documentation also serves itself as text, from `docs.co-relayer.com/llms.txt`, and the whole API as plain Markdown in `/api-reference.md`. Your agent pins the chain id and the contract address from `/.well-known/corelayer.json`, and the SDKs check each relayer against the on-chain list before signing when you pass `verifyRelayer`.

Reading CoRelayer over MCP

An MCP server at `mcp.co-relayer.com` offers a full and a read-only edition, at `/mcp` and `/readonly`. It speaks Streamable HTTP, at protocol revision 2025-06-18. Each tool wraps the API's own routes, with the same authentication and limits, so it adds no capability the API lacks. An agent reads prices with `get_pricing`, network status with `get_network_status`, relayers with `list_relayers` and its quota with `get_quota`.

Buying over HTTP 402

x402 turns HTTP's 402 Payment Required into a real exchange. The API answers with what it wants, the agent signs one payment transaction, and the agent retries. CoRelayer uses it for two purchases, topping up credits and buying a plan; Figure 2 follows a plan. The payment is the same contract call an agent could sign itself, at the same price.

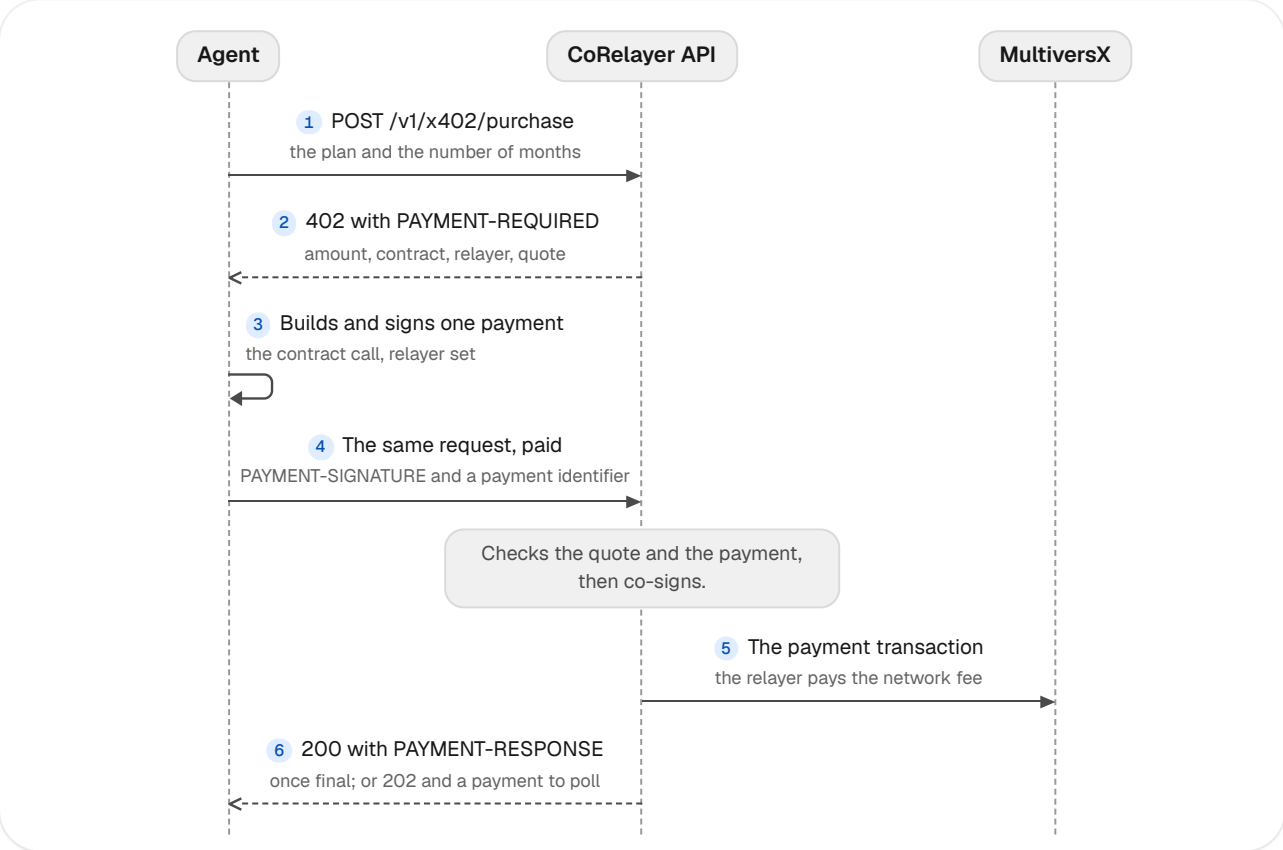


Figure 2 Buying a plan over HTTP 402. The agent signs once, and the API answers once the payment is final.
CoRelayer documentation: x402

The API answers `200` once the payment is final and the purchase recorded, or `202` with a payment to poll at `GET /v1/x402/payments/{paymentId}`. The same payment identifier always gives the same result; `GET /.well-known/x402` and `GET /v1/x402/supported` describe what the API accepts.

> **Pay first, then sign the payload**
>
> A payment and the transaction you want to send both use your account's nonce. Buy, wait until the purchase is final, and only then sign the transaction you came to send. `POST /v1/relay` never takes a payment: its `402` answer carries a `PAYMENT-REQUIRED` header that points to the top-up route.

Compatibility routes

Already on another agent relayer kit? Change one base URL. Three routes answer at the API's root: a health check, a relayer lookup for a sender and a relay endpoint. Like x402 and MCP, they are adapters in front of the same relay pipeline.

09

SDKs

SDKs for TypeScript, Go, Rust and Python. Each wraps the same API, and each language's guide shows code that a test suite runs.

LANGUAGE	NEEDS	GUIDE
TypeScript	Node 20 or later, Deno, Bun or a current browser	docs.co-relayer.com/sdk/javascript
Go	Go 1.24 or later	docs.co-relayer.com/sdk/go
Rust	Rust 1.87 or later	docs.co-relayer.com/sdk/rust
Python	Python 3.11 or later	docs.co-relayer.com/sdk/python

Table 10 The four SDKs, what each needs and where its guide is.
CoRelayer documentation: SDKs

Each SDK in Table 10 does the same things for you:

- It relays a transaction with one signature, and never asks for a second on its own.
- It raises a typed error for every error answer, with the whole problem details document.
- It pays for users from the customer's server with a sponsor key.
- It checks the relayer against the contract through a chosen MultiversX node, when `verifyRelayer` is set.
- It covers every API operation with typed requests and responses.
- It waits for the final state on the intent's event stream, and reads the intent itself only when the stream ends, breaks or stays silent.
- It walks paged lists, checks the signature of a webhook delivery and writes the x402 payment headers.

For another language, generate a client from the OpenAPI document: it describes every route.

10

Where to go next

The reference holds every route, field and code. These pages and contacts are where to go from here.

API reference	docs.co-relayer.com/api/overview
Conventions	docs.co-relayer.com/api/conventions
OpenAPI document	docs.co-relayer.com/openapi.yaml
Error catalogue	docs.co-relayer.com/errors
Quickstart for agents	docs.co-relayer.com/agents/quickstart
Pay for your users	docs.co-relayer.com/sdk/recipes/sponsor-users
Sign and relay	docs.co-relayer.com/sdk/recipes/sign-and-relay
Watch an intent	docs.co-relayer.com/sdk/recipes/watch-an-intent
Verify a relayer	docs.co-relayer.com/concepts/verify-a-relayer
Changelog	docs.co-relayer.com/changelog
Status page	status.co-relayer.com
Plans and prices	co-relayer.com/pricing
Support	support@co-relayer.com
Disclosure policy	docs.co-relayer.com/security/disclosure

Found a problem? Email support@co-relayer.com. Our policy and security.txt say what to include.

11

Glossary and references

Assignment	The answer of `POST /v1/relay/assign`: one relayer in the sender's shard and a lease, valid for 60 seconds, given just before the sender signs.
Chain id	The network identifier inside every signed MultiversX transaction; clients compare it before signing.
Compatibility routes	Three unversioned API routes shaped like a widely used agent relayer interface, so such a kit switches by changing one base URL.
Cursor	An opaque marker of where a page of a list ended; sent back, it returns the next page.
Discovery files	Machine-readable files, such as `llms.txt`, `pricing.json`, `/.well-known/corelayer.json` and the OpenAPI document, that tell software what CoRelayer is, what it costs and where it lives.
Idempotency key	An optional key per user action, sent as `Idempotency-Key` or `intentKey`, that refuses a second transaction for the same action.
Intent	One user-signed transaction CoRelayer was asked to relay, keyed by its sender and nonce.
MCP	The Model Context Protocol, through which AI agents read CoRelayer's prices, network status, relayers and their quota as tools.
Named wallets	Addresses recorded on chain against an account so its plan pays for their transactions; each plan includes a number of them, and each one beyond it carries a small one-time fee.
Native auth	MultiversX's wallet-signature sign-in, used for the dashboard and private reads.
Nonce	The per-sender counter in every transaction; the network accepts each value once, which is why a transaction cannot go through twice.
Problem details	The error format of RFC 9457, a JSON document; CoRelayer's carry a stable `code` and say whether to retry.

Quote	The API's statement of what a transaction or a purchase costs; a purchase quote also gives the ceiling to sign, and is valid for 120 seconds.
Relay Unit (RU)	The API's counting unit: one for every 0.001 EGLD of network fee a transaction could reach at most, so most transfers and contract calls count as one.
Relayer	A MultiversX wallet CoRelayer operates that adds its signature to a user's transaction and pays its network fee, with no authority over any balance.
Settlement	The periodic contract call that debits pay-as-you-go budgets and commits the usage root of its window.
Sponsor key	A secret key on a customer's server that makes the customer's plan pay for transactions any user signs, for the contracts the customer lists.
Usage proof	A Merkle proof that a billed transaction is included in a usage root committed on chain.
x402	An open protocol that turns HTTP's 402 Payment Required into a payment exchange, which agents use to buy credits or a plan without an account.

References

[1] OpenAPI Specification, version 3.1.0. OpenAPI Initiative.
`https://spec.openapis.org/oas/v3.1.0`

[2] RFC 9457, Problem Details for HTTP APIs. IETF.
`https://www.rfc-editor.org/rfc/rfc9457`

[3] x402, an open standard for payments over HTTP 402.
`https://www.x402.org`

[4] Model Context Protocol specification, revision 2025-06-18.
`https://modelcontextprotocol.io/specification/2025-06-18`

[5] MultiversX documentation: relayed transactions.
`https://docs.multiversx.com/developers/relayed-transactions`

[6] MultiversX documentation: native auth.
`https://docs.multiversx.com/sdk-and-tools/sdk-js/cookbook/wallets/native-auth/`

[7] CoRelayer API reference.
`https://docs.co-relayer.com/api/overview`

[8] CoRelayer API, OpenAPI document.
`https://docs.co-relayer.com/openapi.yaml`



CoRelayer

COOPERATIVE RELAYER INFRASTRUCTURE
FOR AUTONOMOUS TRANSACTIONS

co-relayer.com · support@co-relayer.com

© 2026 CoRelayer · CoRelayer API · API overview · Revision 1.0