

Rivus: pay-per-second billing on Tempo

What it is, the problem it solves, how a payment session works, and every step needed to add it to your own service, your own buyer, or your own AI agent.

Version 0.1.0

Network Tempo Testnet (Moderato)

Updated October 2026

01

Overview

Rivus is billing infrastructure that lets a provider of compute or API access charge a customer by the second, or by any other small unit of work, and settle the result on the Tempo blockchain. The customer locks a deposit, pays for each unit as it is delivered, and automatically receives back whatever was not used. Neither side needs an account with the other, an invoice, or trust.

It is built directly on Tempo's Machine Payments Protocol (MPP). Tempo provides the payment channel; Rivus provides the meter: an installable package, `rivus-meter`, with a provider side that times and bills your work, a buyer side that pays for it under a hard ceiling, and a command-line tool that does both from a terminal. The demo in this repository is built on that package.

Who this document is for

Providers who sell compute, inference, data or any API and want to charge for exactly what is consumed. Start at the provider guide.

Buyers who want to pay for metered work from code, with a hard spending ceiling. Start at the buyer guide.

Teams building AI agents, or building with a coding agent, who want the agent to pay for services or to add this billing for them. Start at the agents chapter.

Anyone evaluating the project. The first five chapters explain the idea without any code.

Status

Everything described here runs today on Tempo's Moderato testnet with faucet-issued test tokens. No real funds are involved. Figures quoted as measured come from runs of this project on that network.

02

The problem

Usage of a compute or API service is continuous: a job runs for 23 seconds, a model produces 800 tokens, a feed delivers 40 results. The ways we charge for that usage are not continuous, and each one puts the mismatch on somebody.

How it is billed today	What goes wrong
Flat fee or subscription	The customer pays for capacity whether or not it is used. A job that stops early costs the same as one that runs to the end.
Invoice after use	The provider does the work first and hopes to be paid. That needs a contract, a billing relationship and a collections process before the first request.
Prepaid credits	The customer hands over money in advance. Whatever is not used stays with the provider, and getting it back is a support request.
Card on file	Per-transaction fees make very small charges uneconomic, and a program cannot hold a card.

The mismatch becomes a wall when the customer is software. An autonomous agent can decide in a moment that it needs ten seconds of GPU time or one call to a paid API. It cannot sign a contract, pass a credit check, or dispute an invoice. If agents are going to buy services from each other, payment has to be something a program can do on its own, in the same request as the work, for exactly the amount of work done.

03

The solution

Rivus meters the work as it happens and moves the payment with it. The buyer never pays for more than it received, and the provider never delivers more than it has been paid for.

Exact. Each unit is charged as it is delivered. In the demo a unit of work is billed for the CPU time it actually took, at 0.01 pathUSD per second.

Bounded. The buyer chooses a deposit. That deposit is the most the session can ever cost.

Refundable by construction. When the session closes, the chain pays the provider what was spent and returns the rest to the buyer. Nobody has to ask.

Stoppable. The buyer can stop at any moment and pays only for work already done.

Verifiable. Every session ends in one transaction on Tempo that anyone can look up.

Automatic. The whole exchange happens over HTTP between two programs holding keys. There is no sign-up and no checkout.

What Rivus is not

Rivus does not implement its own payment channel or smart contract. Tempo's protocol already provides that mechanism, and reimplementing it would add risk without adding anything. Rivus is the layer that turns the mechanism into a billing product: what to charge for, when to charge, how to handle a buyer who stops, and how to show both sides what happened.

How a session works

One purchase is one session. A session has four stages. Only the first and the last touch the blockchain; everything in between is ordinary HTTP.

1 Challenge and open

The buyer requests the provider's endpoint. The provider replies with HTTP status 402 (Payment Required) and a challenge that states the price per unit and a suggested deposit. The buyer answers by opening a TIP-20 payment channel on Tempo and locking its deposit in escrow. Nothing has been paid yet. The deposit only sets the ceiling.

2 Stream

The provider starts the work and streams results back as server-sent events. Before each unit is delivered, the provider charges for it. When the amount charged approaches what the buyer has authorised, the buyer signs a new voucher: a small signed message (EIP-712) that says "the provider may claim up to this running total from my deposit". Vouchers are exchanged off-chain, so there is no gas and no waiting for a block between units.

3 Close

When the work is finished, or the buyer stops early, the buyer asks to close. The provider submits one transaction to Tempo that settles the channel for the amount actually spent.

4 Refund

That same close pays the provider and returns the unused part of the deposit to the buyer. The buyer receives a receipt with the amount spent and the transaction hash.

A real session, in numbers

	Amount (pathUSD)
Deposit locked by the buyer	1.000000
Compute delivered	22.73 seconds, 216 tiles
Paid to the provider	0.227300
Refunded to the buyer	0.772700
Network fee paid by the buyer	0.000042
Network fee paid by the provider	0.000049

Spent, not authorised

A buyer can sign a voucher slightly ahead of the work, so the highest voucher can be a little above what was actually charged when a session is stopped mid-unit. The channel settles for the amount spent, and that is the figure Rivus reports. In testing, the reported amount matched the change in both wallets' on-chain balances exactly, after fees.

05

Why Tempo

Tempo is a payments-first blockchain built for stablecoin settlement at high throughput and low cost. Rivus is built on it because Tempo already has, as part of the platform, the pieces that per-second billing needs.

Tempo provides	Rivus uses it for
Machine Payments Protocol (MPP) and the mppx SDK	The whole exchange: the 402 challenge, the session, the vouchers and the receipt.
TIP-20 payment channels	Holding the buyer's deposit in escrow and returning the unused part on close.
EIP-712 signed vouchers	Paying for each unit off-chain, with no gas and no delay between units.
Fees paid in stablecoin	Both parties hold only pathUSD. There is no separate gas token to acquire, which matters for an agent with one wallet.
EVM compatibility	Standard TypeScript tooling (viem) on both sides.

Network details

Setting	Value
Network	Tempo Testnet (Moderato)
Chain ID	42431
RPC	https://rpc.moderato.tempo.xyz
Explorer	https://explore.testnet.tempo.xyz
Payment token	pathUSD, TIP-20, 6 decimals, at <code>0x20c000</code>
Packages	<code>rivus-meter</code> , built on Tempo's <code>mppx</code> (the payment protocol) and <code>viem</code> (chain access)

Quickstart: run the demo

The repository contains a complete, working example: a provider that rents out CPU time to render a fractal image, and a buyer agent that pays for it. Running it takes four commands and no accounts.

Before you start

Node.js 20.9 or later, and npm.

An internet connection, to reach Tempo's testnet RPC and faucet.

Nothing else. No wallet extension, no tokens, no API keys.

Steps

terminal

```
npm install
npm run setup    # rivus init: writes .env.local with testnet-only keys, funds both wallets
                  from the faucet
npm run dev      # the app, at http://localhost:3000
npm run smoke    # second terminal: rivus pay buys a full render and prints the settlement tx
npm run bench    # renders with no payment logic and prints timing
```

1 npm run setup

Generates two fresh testnet keys (provider and buyer agent) and a server secret, writes them to `.env.local` with owner-only permissions, and funds both wallets from Tempo's faucet. It is safe to run again: existing keys are kept and the wallets are topped up.

2 npm run dev

Starts the app. Open `http://localhost:3000` for the overview with the demo embedded, or `http://localhost:3000/demo` for the demo on its own. Press *Start render*.

3 Watch a session

The image fills in tile by tile while the meter counts compute time and cost. Beside it, the balances card shows the buyer's deposit in escrow and the vouchers the provider has received, and the agent traffic pane lists every request the buyer makes as it happens. Let it finish, or press *Stop and refund the rest*. Either way a receipt appears with the amount paid, the amount refunded, and a link to the settlement transaction on the Tempo explorer.

4 npm run smoke

Runs the same purchase from the command line with no browser, using the package's own tool: `rivus pay` opens a channel, pays for a full render, closes, and prints the settlement transaction. Add `-- --stop-after 5` to stop early, or `-- --max-price 0.005` to watch it refuse the price.

```
.env.local (created for you; never commit it)
```

```
PROVIDER_PRIVATE_KEY=0x... # the account that gets paid and signs the on-chain close
AGENT_PRIVATE_KEY=0x...   # the buyer: the account that locks the deposit
RIVUS_SECRET=...          # a random server-side secret; it signs the price quotes the
meter issues
```

07

Provider guide

This chapter adds per-second billing to an endpoint you already have. You write the work; the meter quotes the price, opens the payment channel, times each piece of work, bills it, and settles when either side is done.

1 Install Rivus and create keys

terminal

```
npm install rivus-meter
npx rivus-meter init      # testnet keys in .env.local, both wallets funded from the faucet
```

The provider needs an account on Tempo: it receives the payments and it signs the transaction that closes each channel, so it needs a little pathUSD for fees.

`rivus init` creates one, along with a buyer key for testing and a server secret, and funds both wallets on testnet. It is safe to run again: existing keys are kept.

2 Wrap the endpoint

Create a meter with your price, then hand it the work as an async generator. This is the whole integration. One URL handles every stage of the session: the price quote, the paid stream, the buyer's vouchers and the close.

app/api/render/route.ts

```
import { createMeter } from "rivus-meter/server";

export const runtime = "nodejs";          // long-lived streams need the Node.js runtime
export const dynamic = "force-dynamic";

const meter = createMeter({
  privateKey: process.env.PROVIDER_PRIVATE_KEY, // the account that gets paid
  secret: process.env.RIVUS_SECRET,
  price: "0.01",                               // pathUSD per second of work
  deposit: "1",                                // what a buyer is asked to lock
});

// Write the work as an async generator. Rivus quotes the price, opens the payment channel, times
// each step, bills it, and settles when either side is done.
export const { GET, POST } = meter.route(async function* (job) {
  for await (const result of doTheWork(job.signal)) {
    yield result; // delivered to the buyer, billed for the time it took
  }
});
```

3 Know what gets billed

Each value you `yield` is delivered to the buyer and billed for the time it took to produce, at your price per second. The meter collects payment before it delivers: if the buyer's authorised total is not enough, the stream pauses until a new voucher arrives, so you never hand over unpaid work. Pass `job.signal` into your work so it stops when the buyer leaves.

4 Next.js only: two settings

Load the package at runtime instead of bundling it, and run the route on the Node.js runtime, not the Edge runtime, because the stream is long-lived.

next.config.ts

```
import type { NextConfig } from "next";

const nextConfig: NextConfig = {
  // Load the package from node_modules at runtime instead of bundling it.
  serverExternalPackages: ["rivus-meter"],
};

export default nextConfig;
```

5 Other servers

The meter is not tied to Next.js. It takes a standard `Request` and returns a standard `Response`.

server.ts

```
// Any server that speaks Request and Response: one handler for every method.
const handle = meter.handler(work); // (request: Request) => Promise<Response>
```

6 Test it with a real buyer

Pay your own endpoint from a terminal. The tool prints every exchange, then what was paid, what was refunded, and a link to the settlement transaction.

terminal

```
npx rivus-meter pay https://provider.example/api/render --max 0.5 --max-price 0.02
```

Buyer guide

A buyer is any program with a funded key. Rivus handles the price quote, the channel and every voucher. Your code reads the work as it arrives and closes the session.

1 Install and fund

Install `rivus-meter`. The buyer's wallet needs at least its spending cap in `pathUSD`, plus a fraction of a cent for the fee on opening. On testnet, `npx rivus-meter init` creates and funds one.

2 Open a session, read the work, close

buyer.ts

```
import { createBuyer } from "rivus-meter/client";

const buyer = createBuyer({ privateKey: process.env.AGENT_PRIVATE_KEY });

// Asks the provider for its price, opens the channel and locks the deposit.
const session = await buyer.open("https://provider.example/api/render", {
  maxSpend: "1",           // the most this session can ever cost
  maxPrice: "0.02",       // walk away, before paying, from anything dearer per second
});

for await (const unit of session) {
  use(unit.value);          // one paid piece of work
  console.log(unit.seconds, unit.cost, unit.spent); // what it took, what it cost,
  the running total
}

const receipt = await session.close();          // settles on-chain, refunds the
rest
console.log(receipt.paid, receipt.refunded, receipt.txUrl);
```

3 Set a price limit

The provider states its price before any money moves. With `maxPrice` set, a provider quoting more is refused at that point: `open` throws a `RivusError` with the code `price_too_high`, no channel is opened and nothing is paid.

4

Stop early when you have enough

Leave the loop, call `session.stop()`, or abort a signal. The payments end after the unit in flight; closing the session then settles for the work already done and refunds the rest.

buyer.ts

```
const session = await buyer.open(url, { maxSpend: "1" });

for await (const unit of session) {
  handle(unit.value);
  if (goodEnough()) break;      // leaving the loop stops the payments
}
// From elsewhere: session.stop(), or pass an AbortSignal as { signal } when opening.

// Always close a session that opened: it settles for the work done and returns the
// rest.
await session.close();
```

5

Watch the traffic

`onTraffic` reports every exchange with the provider as it happens, read off the real requests and responses. The agent traffic pane in the demo is this callback, printed.

buyer.ts

```
const session = await buyer.open(url, {
  maxSpend: "1",
  onTraffic: (event) => console.log(event.kind, event.text),
});

// out      GET /api/render
// in       402 Payment Required
// note     price 0.01 pathUSD per compute-second
// out      GET /api/render · open channel
// in       200 stream open · channel 0xaf0e...79e7
// voucher  voucher #1 · 0.010156 · 14 ms
// ...
// in       204 settled on Tempo · tx 0x3903...dd22
```

Field	Meaning
<code>paid</code>	What the session cost, as a decimal pathUSD string. This is the amount the close moved on-chain.
<code>refunded</code>	What came back to the buyer: the deposit less the amount paid.
<code>deposit</code>	What was locked when the session opened.
<code>txUrl</code>	The settlement transaction on the Tempo explorer. <code>txHash</code> is the hash alone.
<code>channelId</code>	The identifier of the payment channel that was used.
<code>outcome</code>	How the stream ended: <code>complete</code> , <code>stopped</code> by the buyer, or <code>budget</code> when the cap was reached.

Two rules for buyers

Set `maxSpend` to the most you are willing to spend on the session: the stream stops when it is reached. And always close a session that opened, including after an error, or the deposit stays locked in the channel.

09

Using Rivus with AI agents

There are two different things people mean by this, and both are supported: an agent that *pays* for services while it works, and a coding agent that *builds* the integration into your product for you.

Your agent as the buyer

An autonomous agent is the customer this kind of billing was designed for. It has no card and signs no contract, but it can hold a key. Give it a wallet with a small balance

and a tool that buys metered work with a ceiling.

1 Give the agent its own wallet

Create a key used only by the agent and fund it with only what you are prepared for it to spend. The balance of that wallet is the agent's total budget, enforced by the chain.

2 Expose one tool

Wrap the buyer in a function your agent framework can call. The agent supplies the URL and a budget for this purchase, and gets back what it spent.

tools/buy-metered-work.ts

```
import { createBuyer } from "rivus-meter/client";

const buyer = createBuyer({ privateKey: process.env.AGENT_PRIVATE_KEY });

// A tool an autonomous agent can call: buy metered work from a URL, never spending
// more than the budget.
export async function buyMeteredWork(options: {
  url: string;
  budget: string; // pathUSD ceiling for this one purchase,
  // e.g. "0.50"
  maxPrice?: string; // refuse providers dearer than this per
  // second
  onUnit: (value: unknown) => void;
  signal?: AbortSignal; // lets the agent stop paying as soon as it
  // has enough
}) {
  const session = await buyer.open(options.url, {
    maxSpend: options.budget,
    maxPrice: options.maxPrice,
    signal: options.signal,
  });

  try {
    for await (const unit of session) options.onUnit(unit.value);
  } finally {
    // Hand the receipt back to the agent as the tool's result.
    const receipt = await session.close();
    return { paid: receipt.paid, refunded: receipt.refunded, proof: receipt.txUrl };
  }
}
```

3

Describe the tool to the model

In the tool description, say what it buys, that it is billed by the unit, that `budget` is a hard ceiling in pathUSD, and that the agent should stop the purchase as soon as it has what it needs. Agents that understand they are paying by the second stop sooner.

4

Keep the key away from the model

The private key lives in the tool's server-side environment. The model only ever sees the tool's inputs and its result. Never place a key in a prompt.

Building with a coding agent

If you build with an AI coding assistant, you can hand it the integration. The two prompts below contain everything it needs and tell it how to prove the result works. Fill in the parts in angle brackets and paste the prompt into your assistant from the root of your project.

Prompt: add per-second billing to my endpoint (provider)

Add pay-per-second billing to an endpoint in this project with the rivus-meter package.

The endpoint to meter: <PATH TO YOUR ROUTE>

Price: <e.g. 0.01> pathUSD per second of work. Suggested deposit: <e.g. 1> pathUSD.

Do the following, in order:

1. Run: `npm install rivus-meter`

2. Run: `npx rivus-meter init`

This writes PROVIDER_PRIVATE_KEY, AGENT_PRIVATE_KEY and RIVUS_SECRET to .env.local and funds both testnet wallets. Make sure .env.local is ignored by git.

3. In the route, import createMeter from "rivus-meter/server" and create one meter:

```
createMeter({ privateKey: process.env.PROVIDER_PRIVATE_KEY, secret:
process.env.RIVUS_SECRET,
price, deposit }).
```

4. Rewrite the endpoint's work as an async generator and export it with

```
export const { GET, POST } = meter.route(async function* (job) { ... }).
```

- Yield each piece of the result as it is ready. Each yielded value is delivered to the buyer and billed for the time it took to produce.

- Pass job.signal into the work so it stops when the buyer leaves.

- If the work measures its own time, yield job.unit(value, { seconds }) to bill that instead.

5. If this is Next.js: set runtime = "nodejs" and dynamic = "force-dynamic" on the route, and add

"rivus-meter" to serverExternalPackages in next.config.

Rules:

- Do not write or deploy a smart contract, and do not call the payment protocol directly. rivus-meter handles the quote, the channel, the vouchers and the settlement.

- Do not guess the API. Read the type definitions in node_modules/rivus-meter/dist before using a function, and stop and tell me if something above does not match them.

- Never commit the keys and never expose them to the browser. Testnet only.

Then prove it works: start the server and run

```
npx rivus-meter pay <THE ENDPOINT URL> --max <DEPOSIT>
```

and show me its output, including the settlement link on the Tempo explorer.

Prompt: let my project pay for metered work (buyer)

Make this project able to pay for metered work with the rivus-meter package.

Provider URL: <THE METERED ENDPOINT>

Spending ceiling per session: <e.g. 1> pathUSD

Highest acceptable price: <e.g. 0.02> pathUSD per second

Do the following:

1. Run: `npm install rivus-meter`
2. Load the buyer's private key from the `AGENT_PRIVATE_KEY` environment variable, server-side only. If there is no key yet, run: `npx rivus-meter init`
3. Create the buyer with `createBuyer({ privateKey })` from "rivus-meter/client".
4. Open a session with `buyer.open(url, { maxSpend, maxPrice, signal })` and read it with `for await (const unit of session)`. Each unit has value, seconds, cost and spent.
5. Support stopping early: leaving the loop, `session.stop()` or aborting the signal must all end the payments.
6. Always call `session.close()` for a session that opened, including after an error, and return `receipt.paid`, `receipt.refunded` and `receipt.txUrl` to the caller.

Rules:

- Never put the private key in client-side code or in the repository.
- Do not guess the API. Read the type definitions in `node_modules/rivus-meter/dist` first.
- Handle `RivusError`: code "price_too_high" means the provider was refused before anything was paid; "insufficient_funds" means the wallet holds less than `maxSpend`.

Then prove it works: run one full session and one that is stopped halfway, and show me that the wallet's balance changed by the amount paid plus a small network fee in each case.

Checking what the assistant built

Ask it to run a session and show you the transaction on the Tempo explorer. A link you can open is better evidence than a passing test.

Compare the receipt's spent amount with the change in the buyer's balance. They should differ only by a network fee of well under 0.0001 pathUSD.

Stop a session halfway and confirm the refund arrives. This is the path most likely to be skipped.

Search the changes for the private key and for `NEXT_PUBLIC`. Neither should appear.

Confirm it did not add a smart contract. If it did, it has misunderstood the task.

Designing your pricing

A meter has one price: pathUSD per second of work. It is stated to the buyer in the price quote before anything is paid. What counts as a second of work can be decided in two ways.

Let Rivus time it

By default the meter clocks how long each step of your generator takes, from the moment it asks for the next value until that value is ready, and bills that. Time spent waiting for the buyer's payment is not counted. This suits work whose cost is the time it occupies: a rented machine, a model call, a long-running query.

Report your own measurement

When the work measures itself more precisely, pass that figure with `job.unit(value, { seconds })` and it is billed instead. The demo does this: the renderer times the CPU work of each tile, so a tile that took 0.31 seconds is charged exactly 0.0031 pathUSD and time parked between tiles is never billed.

```
app/api/render/route.ts
```

```
export const { GET, POST } = meter.route(async function* (job) {
  for await (const tile of renderTiles(undefined, job.signal)) {
    // The renderer times its own CPU work, so bill that figure and not the wall clock.
    yield job.unit(serialize(tile), { seconds: tile.computeMs / 1000 });
  }
});
```

The minimum charge

The first second is paid as the stream opens, which makes 0.01 pathUSD the least a session of the demo can cost. The meter treats that second as credit and draws it down before charging anything more, so the buyer is not charged twice for it.

Choosing a price

The price is yours to set, down to 0.000001 pathUSD per second. A buyer sees it before paying and can refuse it automatically with a price limit, so an endpoint that charges more than its work is worth simply loses the sale, at no cost to the buyer for having asked.

Choosing a deposit

Suggest a deposit that comfortably covers a typical session. The demo suggests 1 pathUSD for a job that costs about 0.23. A deposit that is too small ends the stream early; one that is larger than needed costs the buyer nothing, because the difference comes back on close.

What happens when things go wrong

Situation	What happens
The buyer stops mid-session	The stream ends, the channel closes for the work done, and the rest is refunded. A unit already in progress when the stop arrives may still be charged.
The price is above the buyer's limit	The buyer refuses at the price quote. No channel is opened and nothing is paid.
The buyer's connection drops	In the demo, the buyer agent notices the browser has gone, stops the stream and settles for the work done.
The deposit runs out	The stream stops at the ceiling and the channel settles for the full deposit. The buyer is never charged more than it locked.
The buyer's wallet is underfunded	The buyer checks its balance before opening and reports the shortfall. No channel is opened.
Closing fails once	The buyer retries the close after two seconds before reporting a failure.
The provider fails after the channel opened	The buyer closes the channel and takes the deposit back before reporting the error.
Two sessions from one buyer key	The demo runs one at a time and tells the second caller to try again shortly.

Reference

Environment variables

Name	Used by	Purpose
PROVIDER_PRIVATE_KEY	Provider	Receives payments and signs the close transaction.
AGENT_PRIVATE_KEY	Buyer	Locks the deposit and signs vouchers.
RIVUS_SECRET	Provider	A random secret that signs the price quotes the meter issues.

The command-line tool

Command	What it does
<code>rivus init</code>	Creates any missing keys in <code>.env.local</code> and funds both wallets from the testnet faucet.
<code>rivus pay</code> <code><url></code>	Buys metered work and prints the traffic and the receipt. Options: <code>--max</code> (spending cap, default 1), <code>--max-price</code> (price limit per second), <code>-stop-after</code> (seconds).
<code>rivus</code> <code>balance</code>	Shows both wallets' pathUSD balances.

Routes in the demo

Route	What it does
GET, POST /api/render	The provider's metered endpoint. Answers 402 until paid, then streams tiles.
GET /api/agent? run=ID	The buyer agent. Pays the render endpoint and relays progress to the browser as server-sent events.
POST /api/agent? run=ID	Stops that run. The agent then closes the channel.
GET /api/balances	The on-chain pathUSD balance of both parties.

Events sent to the browser

Event	Contents
started	Both balances as they stood before the channel opened.
tile	One rendered tile, its compute time, and the running total authorised.
log	One line of the buyer's traffic with the provider, as reported by onTraffic.
settled	Amount spent, deposit, the settlement transaction link, and whether the run was stopped.
failed	A message. A settlement can still follow if a channel was open.

Where things live

File	Role
<code>packages/rivus/src/server.ts</code>	The meter: quoting, timing, billing and settling a provider's work.
<code>packages/rivus/src/client.ts</code>	The buyer: price limit, spending cap, traffic reporting, close and refund.
<code>packages/rivus/src/cli.ts</code>	The rivus command-line tool.
<code>app/api/render/route.ts</code>	The demo's metered endpoint, built on the meter.
<code>app/api/agent/route.ts</code>	The demo's buyer agent, built on the buyer.
<code>lib/pricing.ts</code>	The demo's price per compute-second and deposit.
<code>lib/fractal.ts</code>	The compute engine being sold.

Troubleshooting

Symptom	Cause and fix
The metered route returns 500 in Next.js	The bundler tried to bundle the package. Add <code>rivus-meter</code> to <code>serverExternalPackages</code> .
The stream is cut off after a few seconds	The route is on the Edge runtime. Set <code>runtime = "nodejs"</code> .
A "Missing ..." error about a key or the secret	There is no <code>.env.local</code> , or it predates the package. Run <code>npm run setup</code> .
The buyer reports too little pathUSD	Run <code>npm run setup</code> again to top the wallets up from the faucet.
"A render is still running or settling"	The previous session has not finished closing. Wait a few seconds.
Balances look unchanged right after a session	The RPC can lag the settlement by a block or two. Read the balance again.

14

Security and current limits

Testnet only. This build targets Tempo Moderato and uses faucet tokens. It has not been run on mainnet.

Keys stay on the server. Neither key is ever sent to the browser; in the demo the buyer runs server-side for that reason.

The demo generates its keys locally and stores them in a file that is excluded from version control. Use a secrets manager for anything beyond a demo.

Channel state is held in memory. If the provider process restarts during a session, that state is lost. A production provider needs durable storage for it.

One buyer key runs one session at a time in the demo.

The buyer's exposure is limited to its deposit. The provider's exposure is limited to one unit of work, because it charges before it delivers.

No custom smart contract is involved, so there is no contract of ours to audit. The channel logic is Tempo's.

Open-source components

The `rivus-meter` package is built on Tempo's `mppx` for the payment protocol and `viem` for chain access. The demo application uses Next.js with React and Tailwind CSS.

15

Questions and answers

Is the fractal the product?

No. It is a stand-in chosen because it is real CPU work that is easy to watch for half a minute. The product is the meter, which applies to any work that can be billed for the time it takes.

Does every unit cost a transaction fee?

No. Units are paid for with signed vouchers exchanged off-chain. The chain is involved when the channel opens and when it closes. In our runs that cost each side under 0.0001 pathUSD for the whole session.

What stops the provider from claiming more than it delivered?

It can only claim what the buyer has signed for, and the buyer signs as work arrives. If the work stops arriving, the buyer stops signing and closes.

What stops the buyer from taking work and not paying?

The provider charges before it delivers each unit, and the deposit is already locked on-chain. The most a buyer can take without paying is the unit in flight.

Do I need to hold a separate token for gas?

No. On Tempo, fees are paid in the stablecoin itself. Both parties hold only pathUSD.

Can a person use this from a wallet, not a program?

Not in this build. It is designed for keys held by software. A wallet-based buyer would be a different front end on the same protocol.

Glossary

Term	Meaning
Buyer	The party paying for work. Often an autonomous agent.
Provider	The party doing the work and being paid.
Session	One purchase, from opening a channel to closing it.
Payment channel	An on-chain escrow between two parties that lets them exchange many payments off-chain and settle once.
Deposit	What the buyer locks in the channel. The ceiling on what the session can cost.
Voucher	A message signed by the buyer authorising the provider to claim up to a running total.
Settlement	The on-chain close of the channel: the provider is paid and the remainder is refunded.
HTTP 402	The Payment Required status code. The provider uses it to state its price.
MPP	Tempo's Machine Payments Protocol, implemented by the mppx SDK.
TIP-20	Tempo's token standard. pathUSD is a TIP-20 token.
EIP-712	The standard for signing structured data, used for vouchers.
pathUSD	The test stablecoin used for payments on Moderato. It has 6 decimals.
Moderato	Tempo's public testnet.