

FACILITATOR PRESS

x402

THE AGENT ECONOMY

Building and Selling Services
Machines Will Pay For

AGENT · 402 · PAY · 200

FIRST EDITION · 2026

x402 — The Agent Economy

Building and Selling Services Machines Will Pay For

Front Matter

*Published by Facilitator Press. An imprint of Synthex Forge.
First edition, 2026.*

What this book is

This is a build manual for one narrow, unglamorous job: **putting a paid HTTP service on the internet that a software agent can find and pay for, without a human clicking anything.**

That's it. That's the whole book. You will find no grand theory of the coming machine economy here. You can read a hundred essays about it and still not know what to put in your systemd unit file. This book is about the systemd unit file.

Everything in here comes from doing it. We run a live x402 service catalogue on Base mainnet: multiple paid endpoints, real USDC settling on-chain, Ed25519-signed outputs, an MCP wrapper, discovery manifests, the works. We have the transaction hashes. We also have the scars: a leaked origin IP, a near-miss where a broken endpoint almost got listed publicly, and a build pipeline that froze for eight hours because of one orphaned lock file.

Most x402 writing on the internet falls into one of two piles. There is protocol documentation, accurate but stopping at "hello world." And there is breathless speculation about agent economies, entertaining but impossible to ship. This

book is the missing middle. It assumes you can code and know nothing about x402, and it takes you to a listing in a public catalogue.

Who this is for

- **Developers** who keep hearing "agents will pay for APIs" and want to know what that looks like on the wire.
- **Indie hackers and technical founders** looking for a distribution channel that isn't App Store review, isn't SEO, and isn't a sales call.
- **Agent engineers** building tool-using systems who need their tools to be payable, and who are tired of proxying everything through a human-run billing account.

You should be comfortable with an HTTP framework (we use FastAPI), a shell, environment variables, and reading a stack trace. You do *not* need blockchain experience. The parts of blockchain you need are about forty lines of Python, and they're in here.

What this isn't

- **It is not investment advice and it is not a get-rich-quick plan.** Chapter 10 gives you our honest read that this market is early, thin, and mostly pays little right now. If you want a business invoicing \$5,000 a month next quarter, x402 in its current state is not the shortest path. Read Chapter 10 before you read Chapter 4.
- **It is not a protocol specification.** The spec is public and better maintained than any book can be. We cover the mechanics you need to build, and point you at the spec for the exhaustive details.
- **It is not about our trading.** We do run a systematic trading operation. None of it is in this book. The x402 services we describe are general-purpose infrastructure:

document extraction, attestation, provenance. The trading strategies, signals, and proprietary data are deliberately excluded, and we've kept the architecture described here general on purpose so that exclusion cost you nothing.

- **It is not a promise that anything will sell.** We can tell you how to build it, price it, and list it. We cannot tell you that agents will come. Nobody can. Anyone who tells you otherwise is selling something.

An honest note on the market's maturity

You should know what you're walking into.

When we pulled the data for this book, the two public catalogues we could read programmatically carried roughly **2,900 service listings on one side and around 200 on the other**. Combined, the ecosystem reported on the order of **30 million transactions and 20,000 unique buyers** over a trailing 30-day window. That sounds enormous until you look at the distribution. The **median entry price across every marketplace we measured was one cent**, and the bulk of listings clustered between \$0.001 and \$0.03.

Translated: the transaction *counts* are real, the *revenue per call* is tiny, and a great deal of the headline volume is face-value commerce — gift cards, compute purchases — rather than agents buying agent services. A genuine, working, fast payment rail exists here. A thick market of software buyers spending real money on software services does not.

That's the honest framing. It is also, depending on what you want, the opportunity: a rail this good with a market this thin is a rail where a competent builder can still be early. Being early and being paid are different things, and this book will not conflate them.

We flag uncertainty throughout. Where we don't know something, we say so. Where a number comes from our own observation rather than a public source, we label it as such. That discipline is not decoration. It is the reason the rest of the book is worth reading.

CHAPTER 1

Why Agents Are Becoming Buyers

A whole category of software has never needed a billing relationship with anyone. Now it does. That's the entire opportunity, and it starts with what changes when the buyer is a program.

The thing that changed

For thirty years, the buyer of an API has been a human, mediated by an organisation. A developer signs up, a card gets entered, a key gets issued, a quota gets enforced. The key is the unit of identity. The subscription is the unit of payment. Everything downstream — the admin console, the usage dashboard, the seat-based pricing tiers — is scaffolding for the fact that a human occasionally logs in and looks at it.

Now consider the buyer we have. An orchestration loop decomposes a task, discovers that it needs "extract the line items from this invoice," and calls four candidate services. It gets three answers. It picks one.

That buyer never logs in. It never sees a pricing page. It never develops brand loyalty, because it has no memory of brands the way you do — it has a tool description and a response schema.

It cannot use your API-key signup flow, either. Not "won't." *Cannot*. There is no human in the loop to complete the form, and provisioning a key per agent session is a provisioning problem nobody wants at machine speed. When you serve agents, the payment has to happen inline, per call, in the same HTTP round-trip as the request. That's the entire reason x402 exists.

What specifically changes

Four things change, and each one is a design constraint:

1. No cart, no checkout, no account. The payment must be expressible in a single request the agent can construct without a session, a redirect, or a browser. Any flow with a "confirm your email" step is disqualified before it starts.

2. Machine-speed price comparison. If an agent has four candidate services and their prices are machine-readable, it will route to the cheapest one that satisfies its schema — or to the cheapest one whose response it can verify. It has no reason not to. Your marketing copy is not in its context window. Your price is.

3. No branding loyalty, but real trust requirements. An agent doesn't care that you have a nice logo. It cares whether the bytes you returned are the bytes it asked for. Brand equity in the agent economy is provenance, not aesthetics.

4. Idempotency matters more than latency. A human waiting 900ms for an API response will grumble and keep their session. An agent retrying after a timeout will happily pay twice for the same work unless your endpoint makes double-charging impossible. Latency still matters, but at these ticket sizes correctness beats speed.

What does not change

This is where most writing on the topic goes wrong. Agents don't change the fundamental economics of the service business. You still need a thing nobody else does cheaply, you still need it to be reliable, and you still need distribution. The payment rail being novel doesn't make the *product* novel.

An agent buying a generic "summarise this text" endpoint is a commodity buyer, exactly like a human buying the same commodity. The rail lets it transact. It doesn't make the transaction valuable.

The reason to care about x402 is not that it transforms the economics. It's that it *removes a specific friction* — the provisioning step — for a specific class of buyer that previously could not buy anything at all without a human. Removing that friction opens a channel. It doesn't change what you should sell through it.

The thesis, in one line

The public catalogues answer one question loudly: the services earning real money per call are the ones that answer a question a business has to answer — regulatory and government data parsing, company enrichment, attestation, risk assessment. The services with the highest call counts are cheap utilities in a saturated price band, and the incumbents there have better infrastructure than you do.

So the argument of this book fits in a sentence: **compete on verifiability, not on price**. Chapter 10 lays out the catalogue numbers behind that claim, and Chapter 4 turns it into a selection test.

Where the agent-buyer story goes too far

Two claims we don't believe, and won't repeat.

First: "agents will autonomously transact at scale, so volume will be astronomical." Our read of the catalogues says the *infrastructure* for that exists and works, while the *demand* is still mostly human-directed. Agents pay when their operator has funded them and pointed them at something. Fully autonomous agent-to-agent commerce, at the volume the ecosystem's headline numbers imply, is a forecast, not an observation. Treat anyone quoting the transaction counts as proof of the forecast with suspicion — the counts include a lot of non-agent traffic.

Second: "you should build for agents instead of humans." In practice, the money today is overwhelmingly human-directed. The right framing is not "agents instead of humans." It is **"servable to agents and useful to humans."** A service that only makes sense to a machine, with no human-readable value, has no current market. A service that a human finds useful and an agent can also call has two markets.

Where this leaves you

There is a rail that lets a program pay for a service in-band with no provisioning. It works, it's cheap to operate, and the catalogues that list services are public and machine-readable. The market on that rail is real but thin, and the parts of it that are thick are already contested.

So the useful question is not "how do I get in on agent payments." It's this: what can I build that a machine buyer can *verify*, that a machine buyer can't trivially get elsewhere for \$0.001, and that I can list somewhere agents look?

That's the rest of the book.

Next: what x402 actually is, mechanically — including what it does and does not solve.

CHAPTER 2

What x402 Is

Strip away the marketing and x402 is a small, well-specified convention for using a status code that has been sitting unused in the HTTP spec since 1997.

That's it: 402 Payment Required was reserved for future use and then ignored for decades. x402 fills it in, defining *what goes in the response body*, *how the client proves it paid*, and *who settles the transfer*. Everything else — the facilitator, the schemes, the blockchain — exists to make those three questions answerable.

This chapter gives you the mechanics you need to build. The exhaustive version lives in the public specification. Read it once you have something running, because the details matter more when you can see them failing.

The flow, in five steps

The default flow is called the **authorization flow**, and the ordering is verify → resource → settle → respond. Memorise that ordering, because it's what makes the money safe.

1. **Client requests the resource without payment.**

Just a normal HTTP request, with no special headers.

2. **Server responds** 402 with a PaymentRequired object — the "accept-list" of payment options it will accept. Over plain HTTP this object is delivered base64-encoded in a header called PAYMENT-REQUIRED.

3. **Client picks one acceptable option**, constructs a signed `PaymentPayload`, and retries the request with it in the `PAYMENT-SIGNATURE` header.

4. **Verify**. The server (or its facilitator) checks the authorization is valid — *without committing any funds*. This is a read-only check.

5. **Resource executes, then settle**. Only after the work is done does funds movement get broadcast. The server returns a `SettleResponse` in the `PAYMENT-RESPONSE` header.

HTTP 402 flow — how a machine pays for a resource



The authorization flow: the client is refused, pays, and only then receives the resource.

Two consequences of that ordering are worth stating explicitly, because they're the parts people get wrong:

You do not charge for failures. Settlement happens *after* the resource executes. If your handler throws, you never settle. This is the mechanism behind the "failed analyses are not charged" norm you see in established x402 sellers. It's not a policy preference — it's the default ordering. If you build an "upfront" flow (settle first, resource second, declared via `extra.paymentFlow`), you have to handle refunds yourself and you should think hard before doing it.

Verify is not settle. A `verify` call tells you the authorization *would* succeed, and moves nothing. A `settle` call commits. Your architecture should treat them as genuinely different operations, because in the SDK they are.

The three components

Resource server — the thing you're building. It protects the paid endpoints, builds the `PaymentRequired` envelope, and decides whether to serve.

Client — the payer, which constructs the signed payload. In the happy case this is the agent or the agent's library, not your code.

Facilitator — verifies and settles on-chain. This is the piece that means your server never touches the chain directly. You don't need a private key to *receive* money; you need an address. That asymmetry is a genuine operational win, and you should lean on it: your receive wallet can be an address whose key isn't on the serving machine at all.

There are public facilitators (the reference one is at x402.org/facilitator) with `/verify` and `/settle` endpoints. You can also run your own, which we do — it gives you control over the settlement gas and removes a third-party dependency from your critical path. For a first build, use the public one.

The schemas you'll touch

You don't need to memorise all of these. You need to recognise them in a decoded envelope, which is a skill you'll use constantly for debugging.

`PaymentRequired` — what the server sends on 402. Carries:

- `x402Version` (currently 2)

- resource — a ResourceInfo object with url, description, mimeType, serviceName, tags, iconUrl. **This is the object that leaked our origin IP** (Chapter 5, and it deserves its own chapter).
- accepts — an array of PaymentRequirements. This is where price and destination live.
- extensions — free-form. This is where the discovery extension goes, which is how you get auto-indexed (Chapter 7).

PaymentRequirements — one entry in accepts. The fields you must get right:

Field	Meaning	The mistake to avoid
scheme	exact, upto, batch-settlement, auth-capture	Using exact when you mean "up to a budget"
network	CAIP-2 identifier, e.g. eip155:8453	Confusing mainnet 8453 with testnet 84532
amount	Atomic units, as a string	Treating this as a float. USDC has 6 decimals
asset	Token contract address	Wrong-chain contract address
payTo	Recipient address	Sending to your <i>payer</i> wallet by mistake
maxTimeoutSeconds	Payment validity window	Setting it too short and breaking slow agents

The amount field is the one that bites people. USDC on Base has **six** decimals, so one cent is "10000" and one dollar is "1000000". It arrives as a *string*, not a number.

If you parse that string as a float you will eventually drop a fraction of a cent on some amount and not notice. Always `int(amount)` before dividing.

PaymentPayload — what the client sends back. It carries x402Version, the accepted requirements the client chose, and a scheme-specific payload. On EVM the payload contains a signature and an authorization.

SettleResponse — what you return on success: success (boolean), transaction (tx hash), network, payer, amount, plus errorReason when it failed.

EIP-3009 transferWithAuthorization, in plain terms

This is the part that confuses people who haven't done crypto, and it's worth ten minutes of your attention because it explains why the whole thing is cheap.

Normally, to move a token, someone signs a transaction, pays gas, and the transaction lands on-chain — which means the signer needs a native-token balance to pay for gas. That is a real problem for a machine that wants to pay one cent, because the gas would exceed the payment.

EIP-3009 inverts it. The *payer* signs an **authorization**: a message saying "I authorise a transfer of exactly X units of this token, to this address, valid between time A and time B, with nonce N." That signature is created **off-chain**, and somebody else — the facilitator — takes it, submits the actual on-chain transaction, and pays the gas.

Three consequences:

1. **The payer needs no gas.** They need a token balance and the ability to sign — nothing else.
2. **The authorization is bounded.** validAfter and validBefore set a window. A nonce prevents replay. If an authorization is intercepted, it can't be reused and it can't be redirected — the recipient address is inside the signed message.

3. The facilitator needs gas. This is the operational detail that surprises people. Settlement is broadcast by the facilitator's own EOA, which needs a small native-token balance. The *payer's* gas situation is irrelevant, and so is your receive wallet's — your receive wallet never sends anything. We've watched people (including ourselves) lose an hour to this. If payments fail with an insufficient funds for gas error, look at the facilitator's balance, not the payer's.

exact **versus** upto, and why you probably want
exact **first**

exact means exactly the amount in the envelope — a single token transfer for a fixed price. This is what almost every service uses, and what you should use unless you have a specific reason not to.

upto is a pre-authorization with a ceiling: the client authorises up to some maximum, and you settle the actual amount consumed, which may be less. It maps well to metered usage — LLM tokens, compute seconds, variable-size documents.

The reason to start with exact is that upto makes *pricing legibility* harder for the buyer. An agent deciding whether to call you can compare a fixed price against a fixed price. It cannot easily compare "up to \$0.50" against "\$0.05 unless the document is large," and when it can't compare, it tends to route elsewhere. If your costs vary by an order of magnitude with input size, and you can't bound them, upto is the right tool. Otherwise: fixed price, fixed value.

There are also batch-settlement (aggregate many payments into one on-chain settlement — relevant at high call volume where per-call settlement gas dominates) and auth-capture (card-style authorise-then-capture). Neither is where you

start.

Networks and assets

Networks are named by **CAIP-2** identifiers. The ones you'll encounter:

- `eip155:8453` — Base mainnet. **This is where the activity is.**
- `eip155:84532` — Base Sepolia. A testnet, useful, but note that the catalogue activity is on mainnet.
- `eip155:1` — Ethereum mainnet. Expensive gas; not where micropayments live.
- `solana:*`, `tvm:*` and others — the exact scheme has bindings for a long list of chains.

The default asset on USDC-pegged networks is USDC. On Base mainnet, USDC is the contract at `0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913`. Write that down; you will type it many times.

The SDK also accepts a "\$0.10" USD-string syntax that resolves to the right asset and atomic amount for the network. That is convenient, but for anything you're shipping, the explicit string amounts are safer: they can't silently resolve differently after a library upgrade.

Practical recommendation: build on Base mainnet with USDC and the exact scheme. It is where the buyers are, gas is cheap enough that a \$0.02 payment is economically sane, and every tool in the ecosystem assumes it. Test on Base Sepolia if you like, but be aware that testnet success tells you nothing about whether you'll be discovered — the catalogues index mainnet payments.

The HTTP headers, exactly as they go on the wire

Header	Direction	Contents
PAYMENT-REQUIRED	server → client	base64 PaymentRequired
PAYMENT-SIGNATURE	client → server	base64 PaymentPayload
PAYMENT-RESPONSE	server → client	base64 SettleResponse

Two hard-won notes about these.

The 402 envelope lives in the header, not the body. The response body of an unpaid request is typically {}. If you curl your own endpoint and see an empty body, everything is working — you need to decode the PAYMENT-REQUIRED header. This trips up essentially everyone once, including us, and it makes people think their server is broken when it isn't.

Proxies mangle custom headers. You should put a reverse proxy in front of your server, to hide your origin (Chapter 5). You must then verify that PAYMENT-REQUIRED survives byte-for-byte. Some proxies strip unknown headers, some lowercase them, some buffer and rewrite. A status code of 402 through the proxy with a mangled envelope is worse than no proxy at all, because you'll think you're done.

MCP transport

x402 also defines a transport for the Model Context Protocol, and this matters because MCP is how a lot of agents discover tools. The flow differs from HTTP:

1. Client calls a paid tool with no payment.

2. Server returns the tool result with `isError: true` plus the `PaymentRequired` object inside `structuredContent`.
3. Client builds a `PaymentPayload` and **retries the same tool call** with the payment in `_meta["x402/payment"]`.
4. Server verifies, executes, settles, and returns settlement info in `_meta["x402/payment-response"]`.

If you already serve HTTP, wrapping your endpoints as MCP tools is pure packaging — you don't reimplement anything. Chapter 5 covers the wrapper.

There is one specific trap to know now. A stdio MCP probe must send `notifications/initialized` *between* `initialize` and `tools/list`. Miss it and `tools/list` returns an empty array even though the server is fine. If you write a probe and get zero tools, check that before you debug your server.

What x402 does not solve

This is the section that most introductions skip, and it's the most important one in the chapter.

It does not give you distribution. A payment rail is not a marketplace. Enabling x402 gets you a payable endpoint; it gets you approximately zero visitors. This is such a common and expensive misunderstanding that it gets its own chapter (Chapter 3). The short version: x402 routes money, catalogues route attention, and you need both.

It does not verify your output. The protocol guarantees the payment is authentic. It says nothing about whether what you returned is correct, well-formed, or hallucinated. A buyer agent cannot tell a good extraction from a bad one just because the payment verified. That gap is the single largest opportunity in the ecosystem, and it's Chapter 8.

It does not make you discoverable by default. An empty `extensions: {}` in your envelope means the auto-indexing

catalogues will not pick you up at all. Discovery is something you configure deliberately.

It does not handle refunds, disputes, or reputation.

There is no chargeback. There is no dispute resolution. There is no notion of "this seller is bad." Some of that is being built at the edges; none of it is in the protocol.

It does not guarantee the buyer. Settlement happens after the resource executes, which is good for you on failures — but it also means a valid-looking authorization can fail to settle at the last moment. Your handler must tolerate "resource executed, settlement failed." That is an awkward state, and you have to decide how to handle it explicitly.

The mental model to keep

If you remember three things from this chapter, remember these:

1. **Verify before, settle after.** That ordering gives you failure safety for free, at no cost.

2. **Amounts are atomic strings** — six decimals on USDC, and `int()` before dividing.

3. **The envelope is a public advertisement.**

Everything in `PaymentRequired` — including `resource.url` — is visible to anyone who asks your endpoint for a price. Treat it as public data, because it is.

Next: the part that determines whether anyone pays you — discovery.

Get the full book

The rest is the part that costs money to learn the hard way.

x402 — The Agent Economy is 121 pages: choosing a service, building the 402 flow, pricing for machine buyers, getting catalogued, and signing your output so it survives commoditization.

It's written from services live on Base mainnet — including the failures. The IP leak through a reverse proxy. The placeholder that nearly shipped. The eight-hour build deadlock. Those chapters exist because we made those mistakes, not because we read about them.

What's inside

- **12 chapters + 3 appendices** — 121 pages, 25,000 words
- **19 working code blocks** — server skeleton, payment verification, Ed25519 signing
- **Real measured market data** — across ~2,900 services, labelled honestly
- **The pricing chapter** — what machines actually pay, from \$0.001 to \$2.00
- **A launch checklist** and a glossary

Get it

\$29 — instant PDF download, DRM-free, yours to keep.

<https://synthexforge.com/ebook>

If it doesn't teach you something you can act on, ask for a refund.