ChatGPT

Building an x402 Solana SDK for Autonomous Agent Payments

Introduction:
x402 is an emerging open protocol that brings the long-reserved HTTP 402 "Payment Required" status code to life, enabling pay-per-use access to APIs and web services via blockchain micropayments [1] [2]. It was co-developed by Coinbase and Cloudflare to facilitate machine-to-machine and AI agent commerce without traditional accounts or subscriptions [3]. In a typical x402 flow, a client (e.g. an AI agent) requests a resource and receives an HTTP 402 response containing payment details (amount, token, recipient). The client then executes the required on-chain payment and re-sends the request with proof of payment in an `X-Payment` header, after which the server provides access [4]. Solana is a popular blockchain for x402 because of its high throughput, ~400ms finality, and negligible fees (~$0.00025 per txn) which make frequent micropayments economically viable [5]. With native USDC stablecoin support on Solana, pricing for services can remain stable and predictable globally [6] – a key factor for AI agents transacting autonomously. Building a dedicated **x402-Solana SDK** will empower developers to easily integrate this payment flow into applications and AI agents. Such an SDK should abstract the complexity of Solana transactions and HTTP 402 handling, enabling "autonomous payments" as a seamless part of web requests. Below, we explore what is needed to create this SDK, including language choices, existing SDK references, and the design of a solution that handles **"most things (if not all) to do with x402 and agentic functions."**

Language and Environment Considerations

Choosing a Language: Solana's on-chain programs (smart contracts) are written in low-level languages like Rust (and C/C++ via LLVM) [7]. However, when building an off-chain SDK (for clients/servers), the ecosystem leans heavily toward higher-level languages. In particular, **TypeScript/JavaScript is the most widely used language for Solana development off-chain**, thanks to official libraries like Solana Web3.js and the new Solana Kit SDK [8]. In fact, `@solana/kit` is now the recommended Solana SDK and is implemented in TypeScript [8]. This reflects how Solana developers commonly use TypeScript/JS for writing dApps and backend integrations. For instance, Solana Web3.js (the legacy JS library) is a TypeScript library that can run in Node.js, web browsers, and even React Native, making it very versatile [9]. Given this landscape, building the x402 SDK in **TypeScript** is a sensible choice. It aligns with the tools and languages most Solana developers know, and it allows the SDK to be used in both Node.js environments and in web front-ends (via bundlers or frameworks) with the same codebase.

Why Not Java (or Others)? The user suggested considering Java, as Java is popular in some enterprise contexts. There is indeed interest in Java for x402 – for example, the Mogami project is developing a Java-based x402 SDK and related tools [10]. Java could be beneficial for enterprise servers or Android-based agents. However, Java is currently far less common in the Solana community compared to JavaScript/TypeScript. The Solana Foundation's official client SDKs prioritize Rust, TypeScript, and to a lesser extent Python [8] [11]. Thus, **TypeScript** will likely reach the largest developer audience initially. That said, designing the SDK with clean abstraction could enable implementing other language versions in the future

(e.g. a Java port) if demand arises. Python is another possible language (Coinbase's reference x402 implementation is in Python [12]), but again, for Solana integration, TS/JS has the richest support and libraries. In summary, **TypeScript (Node.js)** is the primary target language for the SDK, with the option to later create separate SDKs for other languages if needed ("create differently for each of them," as the user notes).

Node.js and Browser Compatibility: It's important that the SDK works both in Node.js (server or backend agent context) and in browser environments. Node.js support is crucial for server-side applications or autonomous agents running in cloud environments. Browser support enables integration directly into web apps or browser-based agents. Fortunately, a TypeScript codebase can usually be made isomorphic (usable in both environments). Solana's TS libraries are built with this in mind – for example, Solana Web3.js can run in the browser (using web wallets or a provided keypair) just as well as in Node.js [9]. To achieve cross-environment compatibility, the SDK should use APIs that exist in both worlds. For instance, using the standard `fetch` API (or a polyfill) for HTTP requests will work in browsers and Node (Node 18+ has `fetch` built-in, or one can use a polyfill like `node-fetch`). Likewise, cryptography and base64 encoding in JavaScript are available via built-in APIs or libraries usable on both platforms. One consideration is **key management**: in Node, an agent SDK might load a private key (for signing transactions) from a secure store or environment variable. In a browser, holding raw private keys is risky; typically, web apps rely on wallet extensions (e.g. Phantom for Solana) to sign transactions. Our SDK should accommodate both scenarios. It could allow injection of a custom signing function or wallet interface, so that in a browser context the SDK can defer to an external wallet (with user approval), whereas in Node it might directly use a provided `Keypair` (for fully autonomous operation). Overall, focusing on TypeScript/Node gives maximum flexibility – it's a language **most Solana devs use** and it naturally supports both server and browser use cases.

Learning from Existing x402 SDKs and Tools

Before building a new x402-Solana SDK, it's wise to study the landscape of existing SDKs and reference implementations. As of late 2025, x402 is very new but a number of projects and companies are already developing tooling around it [13]. According to Solana's documentation, *"the x402 community spans 40+ partners building everything from client SDKs to payment facilitators."* [13]. However, the space is evolving rapidly – *"At the moment it is not yet clear which of the 402 SDKs will be the most popular."* [14]. This means we should draw inspiration from multiple sources:

- **Coinbase Reference Implementation:** Coinbase, as a co-creator of x402, has an open-source reference implementation of the protocol [12]. Their public repo "coinbase/x402" provides the core specifications and example code (currently with Python and some JavaScript). This is a fundamental resource for understanding the **spec** – e.g., the JSON structures for payment requests and the workflows for verifying payments. The Coinbase implementation defines the *scheme* concept (currently `exact` payments are implemented, with `upto` proposed for usage-based billing) [15]. Our SDK will adhere to these specs. For example, the `PaymentRequirements` JSON (sent in the 402 response) and the `X-Payment` header contents should match Coinbase's standard. Coinbase's reference also includes a facilitator API (/verify, /settle endpoints) [16] which can optionally be used to offload some blockchain logic to a third-party service – our SDK might not need to run a facilitator, but could be compatible with one if needed.

- **Corbits:** Corbits is described as a *"Solana-first SDK to implement x402 flows quickly on Solana."* [17] . This suggests Corbits is very relevant, likely implemented in TypeScript or Rust with a focus on Solana's token standards. Corbits provides a high-level API for x402; studying its documentation (at **corbits.dev**) would show how they handle things like token transfers, associated token accounts, and network selection in a user-friendly way. We know Corbits supports Solana fully and likely simplifies the client-side process (possibly even automating the HTTP calls and transaction creation). By reviewing Corbits, we can glean best practices for API design – e.g., how to represent the payment request, how to expose a single call to "fetch with payment" versus multi-step calls. If Corbits is open-source, its code could inspire our implementation details as well.

- **Thirdweb Nexus:** Thirdweb (a popular web3 developer platform) has announced support for x402 in their SDK, via a product called Nexus [18] . They emphasize enabling developers to monetize backend services with seamless payments, and even mention a *"gasless Facilitator wallet supporting 80+ chains"* [19] . This indicates Thirdweb's approach might abstract away the transaction signing from the client by using a relay/facilitator. While our focus is Solana specifically, thirdweb's model reminds us of the possibility of **gasless** or delegated transactions – e.g., an SDK could optionally submit the payment transaction through a proxy service to avoid requiring the client to hold SOL for fees. For now, we'll design with the simpler model (client signs and pays directly), but keeping an eye on how thirdweb achieves multi-chain support and gasless experience could inform future enhancements.

- **Cloudflare Agents & Coinbase AgentKit:** Cloudflare's *Agents SDK* and Coinbase's *AgentKit* are frameworks for building AI agents, and both *"natively support x402"* [3] . This means an AI agent built with those tools can automatically handle 402 responses and perform payments. While those are more high-level platforms (Cloudflare's is likely integrated into their Workers environment, and AgentKit might be a Python library), it's useful to consider their features. For instance, Cloudflare might provide an event-loop or middleware where any 402 response triggers a callback for payment. Our SDK could similarly offer an **agent-friendly interface**, such as a wrapper around `fetch` that automatically performs the x402 handshake. The key takeaway is that automation and simplicity are crucial – an AI agent developer should be able to enable x402 payments with minimal code (perhaps one function call or a plugin into their agent framework).

- **Mogami (Java SDK):** As noted, Mogami is focusing on a Java-based x402 stack [10] . Their existence confirms that Java support is considered important by some, but also provides architectural insight. Mogami's offerings include an "x402 Facility Server" and console [20] , suggesting a clear separation between client SDK and server-side payment facilitator. We can mirror this idea by structuring our SDK into client and server components (more on this below). If possible, reviewing Mogami's documentation could reveal how they structure their SDK classes and methods in an OOP context, which can inspire analogous structure in our TS SDK (even if TypeScript tends to be more functional). Mogami's focus on *tools and services* around x402 implies our SDK might not just be a few functions, but potentially part of a larger toolkit (CLI tools, monitoring, etc.) – we'll scope our initial build to core functionality but keep extensibility in mind.

- **Other x402 Projects:** The Solana guide lists additional projects like **MCPay.tech** (for Model Compute Protocol payments) [21] , **PayAI** facilitator [21] , **ACK** (Agent Crypto Kit by Automation – which includes x402 support) [12] , **Crossmint**, and Google's A2A x402 example [22] . Each of these has a specific angle (e.g., ACK likely issues JWTs post-payment for session reuse [23] , MCPay focuses on AI model agent payments, etc.). For thoroughness, we would glance at their docs to see if they solved any

tricky aspects. For example, ACK using JWTs suggests a feature where after a successful payment, the server issues a short-lived token so the client doesn't have to pay again for every request in a timeframe [24]. This is an enhancement we might consider supporting (like our SDK could optionally handle storing and sending such a token). These existing solutions collectively guide us on **what features developers value**.

Design Goals and Key Features of the SDK

To *"handle most things (if not all) that have to do with x402 and agentic functions,"* our SDK design will cover the full lifecycle of an x402 payment transaction, from initial request to payment execution and verification. The major goals and features include:

- **1. Payment Request Handling (HTTP 402 Detection):** The SDK should make it easy to detect when a resource requires payment and retrieve the payment terms. This involves interpreting the 402 response from a server. Typically, the server's response will include a JSON body (often called `PaymentRequirements`) detailing the required payment: which token (mint address) or currency, the amount, the recipient's address (or token account), and possibly a message or other metadata [25] [26]. The SDK could provide a method like `fetchWithPayment()` or a middleware that does: first attempt to GET the resource, and if a 402 is encountered, automatically parse the quote. The PaymentRequirements structure is standardized by the x402 spec. For example, on Solana it might look like: `{ payment: { recipientWallet, tokenAccount, mint, amount, amountUSDC, cluster, message } }` [25] [26]. Our SDK will parse this and expose it as a friendly object (e.g. a TypeScript interface).

- **2. Wallet and Token Management:** To pay the required amount, the client (agent) needs a Solana wallet with the appropriate token. The SDK should integrate with Solana's wallet standards. In a Node context, the developer might supply a `Keypair` (private key) for the agent's wallet. In a browser, they might use a wallet adapter or our SDK could prompt for a wallet connection. We will make wallet handling modular. Additionally, the client must have an **associated token account** for the token (like USDC) to spend from. If the agent's wallet doesn't already have an associated token account for, say, USDC, the SDK can automatically create it (using Solana's SPL Token library). The server typically provides the **recipient token account** (the merchant's account to receive the payment) in the quote [27] [28]. However, if for some reason the provided token account doesn't exist on-chain, the SDK could detect that via an RPC call and create it (though in practice, the server should ensure their receiving account exists; our example code still checked and created it if needed [29] [30]). In summary, our SDK will handle token accounts seamlessly: ensuring the payer's sending account exists and has sufficient balance, and validating the receiver's account.

- **3. Transaction Construction:** The core of the payment is constructing a Solana transaction that transfers the specified amount of the token from the agent to the resource provider. Using Solana's libraries (`@solana/web3.js` and `@solana/spl-token`), the SDK will create a new `Transaction` object. If needed, it will add an instruction to create the associated token account for the recipient (as shown in the Solana guide's minimal example [30] [31]). Then it adds the **SPL Token transfer** instruction to move the required number of tokens from the payer to the recipient [32] [33]. The amount is specified in the smallest unit (e.g. for USDC with 6 decimals, an amount of 100 means 0.0001 USDC in the example [34] [35]). Our SDK will likely provide a helper to convert human-readable token amounts to base units and vice versa, or simply require the server-provided

amount (already likely an integer of base units) to be used directly. After adding instructions, the SDK will **sign the transaction** with the agent's private key (or via wallet adapter). Note that in an x402 flow, the client *does not immediately broadcast the transaction* – instead, it will package it and send to the server, so the transaction is signed but **not yet submitted on-chain by the client** [36] [37].

- **4. X-Payment Header Encoding:** According to the x402 spec, the client must send proof of payment in an `X-Payment` HTTP header on the follow-up request. Our SDK will automate creation of this header. The header's value is a base64-encoded JSON string containing fields: `x402Version` (currently 1), the payment `scheme` (e.g. `"exact"` for a fixed price payment), the `network` (which would be `"solana-mainnet"` or `"solana-devnet"` etc., as specified by the server's quote), and a `payload` object. For the `exact` scheme on Solana, the payload typically contains the `serializedTransaction` (base64 representation of the signed transaction) [38]. In other words, the client signs a transaction paying the required amount to the given account, then encodes that signed transaction to include in the header. Our SDK can handle all these steps with one call. For example, after constructing `tx`, we do something like: `const serializedTx = tx.serialize().toString('base64')`, then build the payment proof JSON and base64-encode it [39] [40]. This header format is standardized (the Coinbase repo defines it clearly). The SDK will ensure the JSON is correctly structured and encoded.

- **5. Retrying the Request with Payment:** Once the X-Payment header is prepared, the SDK will retry the original HTTP request (GET/POST to the protected resource) with this header attached. This can be abstracted as part of `fetchWithPayment()` or similar. Essentially, the SDK should handle the loop: *make request -> if 402, pay -> make request again with proof -> return result*. From the developer's perspective, using the SDK might look like: `const result = await x402Client.get(url, options)` and under the hood it does the 402 flow. We will of course also expose lower-level methods for more control if needed (like separate `getPaymentQuote()`, `pay()` etc., in case a developer wants to handle things manually or do custom logic between steps).

- **6. Server-Side Verification (for completeness):** While the main focus is on the client/agent side (since the question leans towards agentic functions), a full-featured SDK could also provide utilities for the *server side* (the API provider) to simplify their implementation. On the server, implementing x402 involves: returning a 402 with payment info, and if a payment proof is received, decoding and verifying it. Our SDK could include a server module (for Node/Express, for example) that performs this verification automatically. The verification process on Solana is multi-step:

  - Decode the base64 X-Payment header back into a JSON object [41] [42].
  - Parse the serialized transaction bytes back into a `Transaction` object [43] [44].
  - Check that the transaction contains a valid token transfer instruction to the expected recipient account for at least the required amount [45] [46]. (Our example code loops through instructions to find a Token Program transfer and validates the destination matches the quoted `RECIPIENT_TOKEN_ACCOUNT` and amount ≥ price [47] [46]).
  - Optionally simulate the transaction on Solana (using `simulateTransaction`) to ensure it would succeed (no errors) before submitting [48] [49].
  - Submit the transaction to the Solana network (e.g. via `sendRawTransaction`) and wait for confirmation [50] [51].
  - Once confirmed, double-check the post-transaction token balances to confirm the exact amount received by the recipient (guarding against any edge cases) [52] [53].

- Finally, send back a 200 response with the protected content, and possibly some receipt info (the server example returns the transaction signature and an explorer URL along with the data) [54] [55].

Our SDK can wrap much of this server logic into a convenient function or middleware. For example, an Express middleware could handle the above automatically: if `X-Payment` is present, do verify & settle; if not present, respond with 402 and payment requirements. This would greatly simplify adoption for API providers. Since the question is mostly about building the SDK (not just using one), including server utilities would make our SDK more comprehensive. It aligns with the idea of handling "all that has to do with x402" – both client and server sides of the transaction.

- **7. Agent-Friendly Abstractions:** Given x402's primary use-case is *autonomous AI agents*, the SDK should be designed for easy integration into agent workflows. In practice, an AI agent might use a framework (LangChain, Cloudflare Workers, Coinbase AgentKit, etc.) and making an HTTP call is just one step in its reasoning loop. We want to minimize the friction to add payment capability. Some ideas: We can provide a **wrapper around** `fetch` that the agent can use to automatically handle 402 flows. For example, a drop-in replacement function `fetchX402(url, options)` that internally does what we described (checking for 402 and handling payments). This way, an agent developer doesn't have to write the boilerplate – they might simply swap their normal fetch with ours when accessing certain endpoints. Another approach is offering a **callback or event system**: for instance, if using an agent framework, the SDK could expose hooks like `onPaymentRequired(url, paymentInfo)` that the agent can register to decide whether to pay (perhaps an agent might want to apply some policy, e.g. only pay up to a certain amount or only for certain resources). However, in many cases the agent will be programmed to pay automatically, so the default behavior is just pay and proceed.

- **8. Multi-Token and Network Support:** While most x402 examples use USDC (a stablecoin) on Solana, the protocol is chain-agnostic and supports any token depending on the server's request [56]. Our SDK should not be hard-coded to USDC; it should work with any SPL token mint and allow payments on devnet, testnet, or mainnet. The `network` field in the X-Payment header ensures that the server knows which chain the payment is on (e.g., `"solana-devnet"` vs `"solana-mainnet"`) [57]. So, the SDK should map Solana cluster endpoints to the appropriate network identifiers. It should also perhaps allow the developer to configure which RPC endpoint or connection to use (for example, default to Solana's public RPC or a custom RPC node via `Connection` object). Additionally, if the server's quote allows multiple options (imagine a server could accept payment on Solana or another chain), the SDK might allow choosing one – but this is an edge case; typically a given endpoint will specify one network/token for payment. The key is flexibility: support all SPL tokens and cluster types.

- **9. Security and Reliability:** The SDK must handle errors and edge cases robustly. For example, if the agent's token balance is insufficient, the SDK should catch that and perhaps throw a descriptive error (the example client code explicitly checks balance and errors out before proceeding [58]). If the on-chain transaction fails (e.g., network issues, or the server fails to broadcast it), the SDK should report that back to the caller. Also, careful handling of sensitive data: if using a private key, ensure it's never inadvertently logged or exposed. In a browser context, we avoid storing keys in plaintext; we rely on secure wallet providers. The SDK can encourage best practices by requiring keys via environment variables or parameters, not hardcoded in code. We should also follow Solana's security patterns, such as using recent blockhash for transactions and proper confirmation strategy (the example uses

`confirmTransaction` with commitment "confirmed" [59] [60]). Duplicate transaction submissions should be handled gracefully (Solana automatically rejects duplicate sigs [61], but our client could also use unique recent blockhash each attempt). Essentially, the SDK should aim to make the payment flow **as safe and idempotent as possible**, abstracting the nuances away from developers.

- **10. Developer Experience:** Lastly, a general goal is to make the SDK intuitive. This means good naming (e.g., a class `X402Client` with methods like `requestResource()` or `payAndRequest()`), thorough documentation, and possibly providing TypeScript types for all the structures (PaymentRequirements, PaymentProof, etc.). If we cover "most things" about x402, we might even include tools like a small CLI to inspect transactions or to generate a payment request JSON for testing, etc. But those are nice-to-haves; the core is providing a clean API such that with a few lines of code, a dev can enable x402 payments in their app. Given the complexity under the hood, achieving a one-liner integration (akin to *"one function call for the client"* as some x402 promos describe [62]) is a worthy goal.

Figure: High-level x402 payment flow. The client requests a resource and gets a 402 Payment Required with details. The client then submits the required payment on Solana (often via a facilitator or by sending the transaction to the server in the X-Payment header) and upon verification, the server returns the content. [4] [3]

Architecture and Implementation Plan

With the goals outlined, we can architect the x402 Solana SDK in two main parts: a **Client Module** for agents/consumers and a **Server Module** for resource providers. Each will leverage Solana's libraries and handle the respective side of the protocol.

Client Module (Agent SDK): This will be the core of our SDK, likely implemented as a class or set of functions in TypeScript. Key components and steps:

- *Configuration:* The client needs to know what Solana connection to use (devnet/mainnet, custom RPC URL or default) and have access to a signer (private key or wallet interface). We'll provide a way to initialize the SDK with a config object containing these. For example: `const x402Client = new X402Client({ connection, payerKeypair })` or `payerWallet` (abstract interface for either Keypair or a wallet adapter). If using Node, `payerKeypair` can be loaded from a JSON file or env variable (as shown in the Solana example where they read `client.json` with a secret key [63]). If using in a browser with a wallet extension, the user of our SDK might supply a function like `signTransaction(tx)` instead.

- *Making Requests:* The primary API could be a method like `x402Client.fetch(url, options)` (mirroring the Fetch API) or specialized methods for GET/POST etc. Internally, this method will perform the logic: do an HTTP request via `fetch`. If the response status is 402, parse the JSON for payment info. For example, after `const response = await fetch(url, options)`, if `response.status === 402`, then do `const paymentRequirements = await response.json()`. The SDK should validate this object – ensure it has the fields we expect (`mint`, `amount`, etc.) [64] [65] and that the `cluster` or network matches our client's configuration (e.g., if the server expects devnet but our connection is to devnet, good – if mismatch, we might need to handle that or error out). Next, we construct the transaction for payment: use `mint = new`

`PublicKey(payment.mint)` and `tokenAccount = new PublicKey(payment.tokenAccount)` as given by server [27] [66]. We then ensure the payer has an associated token account for that mint (using `getOrCreateAssociatedTokenAccount` from `@solana/spl-token` [67] [68]). We check the balance with `connection.getTokenAccountBalance(payerTokenAccount)` [69] and compare to `amount` [58], throwing an error if insufficient (the developer can catch this to inform that the agent's wallet needs funds). Assuming sufficient balance, we proceed to build the transaction. We use `Connection.getLatestBlockhash()` to get a recent blockhash for the transaction [70]. If the server-provided recipient token account might not exist (the example checked by attempting `getAccount` on it [71]), we handle that: if not exists, create an ATA via `createAssociatedTokenAccountInstruction` (the example hardcoded the recipient's owner wallet for demo [72] [73]; our SDK would ideally use a recipient wallet field if provided – the example's JSON didn't explicitly send recipient's base wallet, but in code they had it as `RECIPIENT_WALLET`). In many cases, the server will have pre-created its token account, so this step is rarely needed; still, we implement it for completeness. Then we add the `createTransferInstruction` to transfer `amount` tokens from payer's ATA to recipient ATA [32] [33]. We sign the transaction with the payer's key. **Important:** We do *not* send this transaction via RPC ourselves. Instead, we serialize it to base64. We form the `X-Payment` header as described earlier: JSON with `scheme: "exact"`, `network: "<solana-network>"` (e.g. `"solana-devnet"`), and `payload.serializedTransaction = <base64_tx>` [38]. We base64-encode this JSON string to produce the header value [40]. Now we retry the request: `await fetch(url, { ...options, headers: { "X-Payment": headerValue } })`.

The response to this second request should be a 200 (if payment is verified and successful) with the actual content. Our function will then return that content (or the full Response object). If the response is still 402 or an error (maybe the payment failed verification), we should surface that (perhaps throw an `X402PaymentError` with details from the response JSON). In the example server, if verification fails (say the transaction was wrong amount), they return 402 with an error message [74] [75], which our client can catch and propagate. Assuming success, the server might include some extra info like a `paymentDetails` or a confirmation that we might choose to log or handle (the example returns a JSON with `data` and `paymentDetails.signature` etc. [76] [77]). Our SDK could optionally parse that and, for instance, log the explorer URL for the transaction (helpful for debugging) [78]. But the main output is the protected resource data.

All these steps will be wrapped so that developers don't have to write them each time. Essentially, the **client module** implementation is an automation of the manual flow shown in Solana's guide (Minimal Client code) [79] [80]. By studying that code (which we have), we ensure our implementation covers each step.

- *Edge case handling:* If multiple consecutive payments are needed (unlikely for a single request, but if an agent is interacting with many endpoints, it might face multiple 402 responses), the SDK handles each independently. If a server returns a *JWT token* after a payment (like ACK does) [24], our client could store it (maybe in a cache or in-memory) and automatically use it for subsequent requests to the same service (to avoid paying repeatedly within some time window). This would be an advanced feature – perhaps configurable (e.g., `enableSessionToken: true/false`). Initially, we may note it as a possible improvement (as the Solana guide itself suggests using a JWT for reuse [24]).

- *Testing:* We should plan for testing the client. We can run it against a known x402-enabled service (perhaps a local Express server using our server module or the minimal example server). Writing unit tests for pieces (like ensuring the header encoding/decoding is correct) is also important. The Coinbase repo's test cases (if any) could be borrowed to validate compliance with the spec.

Server Module (Provider SDK): This part of the SDK targets developers who operate the paid API or resource. While not explicitly asked, it completes the picture to handle "all things x402." The server module could include:

- *Middleware or Helper Functions:* For an Express.js server, for example, we can provide a middleware like `x402.requirePayment(paymentOptions)` that can wrap an endpoint. The developer would specify what payment is required (e.g. `paymentOptions = { amount: 100, mint: USDC_MINT, cluster: 'devnet' }`). When a request hits that endpoint, the middleware would check for `X-Payment` header. If none is provided, it responds with status 402 and a JSON containing the PaymentRequirements (the code would fill in `recipientWallet` or `tokenAccount`, etc., based on either configured values or on-chain derivation) [25] [26]. If an `X-Payment` header is present, the middleware will decode and verify it as described earlier (introspecting the transaction). Our server utility can reuse a lot of code from the Solana example: checking the transaction's instructions for a valid SPL token transfer to the expected account [47] [46], simulating it [48] [81], then sending it to the network and confirming [50] [82]. After confirmation, it should verify the token balance delta (to be absolutely sure the correct amount was received) [83] [53]. Only then does it proceed to call the actual handler to produce the resource (or it can directly send a response if the resource is static). If any step fails, it returns 402 with an error message so the client knows it didn't go through. Essentially, our middleware would encapsulate the logic shown in the minimal Express server code [84] [42] and make it reusable with one line integration.

- *Integration with Facilitators:* Some providers might use an external facilitator service instead of directly interacting with the blockchain. The x402 spec defines optional `/verify` and `/settle` endpoints that a facilitator can expose [16]. For instance, a server could forward the payment proof to a facilitator API, which would handle checking the chain and returning a yes/no. Our SDK could support this by offering a mode where the developer supplies a facilitator base URL. In that mode, when `X-Payment` is received, we'd call the facilitator's `/verify` endpoint with the payload, rather than connecting to Solana directly. This can simplify the server's responsibilities (no need for a full RPC client locally). However, using a facilitator introduces an external dependency and possibly fees or trust considerations. Initially, we can implement direct on-chain verification (since that's straightforward with Solana's JS API), and leave hooks for facilitator integration (perhaps a function option to override the verify behavior).

- *Multiple Payment Options:* In advanced scenarios, a server might accept payment on multiple chains or tokens (e.g., either USDC on Solana or ETH on Base). The PaymentRequirements spec can list multiple options. Our server SDK could help by generating that structure. However, handling multi-chain is complex (it means the client picks one option to pay). In a first iteration, we might assume a single required payment option for simplicity. The SDK could be extended later to allow an array of acceptable payments and to verify accordingly.

- *Logging and Monitoring:* The server module could log payment attempts, successes, and failures, which is useful for developers to monitor usage. For example, logging the transaction signature and amount received (the example server prints details to console on success [85]). Our SDK could expose events or simply use a logging mechanism for these.

Both client and server modules would be part of the same SDK package for convenience, or possibly split into `x402-solana-client` and `x402-solana-server` packages if we want to keep dependencies lean (e.g., a front-end might not need server code). Since we are focusing on Node/TS, one package with conditional exports may suffice.

Implementation Libraries and Requirements

To build this SDK, we will leverage existing Solana libraries and some general utilities:

- **Solana Web3.js / Solana Kit:** We will use the Solana JavaScript API for all blockchain interactions. This includes creating `Connection` to a Solana RPC node (for sending transactions and querying balances) and constructing transactions/instructions. The code in our SDK will mirror what a developer would do manually with these libraries. For SPL token operations (associated accounts, token transfers), we'll use `@solana/spl-token` which provides convenient methods like `getOrCreateAssociatedTokenAccount` [67] and `createTransferInstruction` [32]. These battle-tested libraries save us from dealing with low-level details (like token program IDs, instruction layouts, etc. – although our server verification does peek into the instruction binary to verify it's a transfer [86]). Notably, Solana's new SDK "Solana Kit" might also offer higher-level constructs (depending on its release status) – but since Web3.js is known and reliable, we can start with that.

- **HTTP Library:** We plan to use the standard `fetch` API for making HTTP requests, to maintain cross-platform compatibility. In Node, if needed, we ensure a global fetch is available (`node-fetch` or just require Node >=18). We might also consider using Axios or another library if we need advanced features, but fetch should suffice and keeps dependencies minimal.

- **Encoding and Crypto:** For base64 encoding and decoding, in Node we have `Buffer` built-in, and in browser `btoa/atob` or `TextEncoder`. We'll likely use Buffer in our Node code (as in the examples [87] [43]) and ensure it's polyfilled or replaced in browser builds. JSON handling is straightforward with `JSON.parse` / `JSON.stringify` (as seen in example code on both client and server side [87]). We should also consider using a cryptographic library if we need to verify signatures manually, but since we rely on `Transaction.from()` to parse and Solana RPC to confirm, we may not need manual signature verification (the RPC confirmation suffices).

- **Stablecoin/Tokens metadata:** Our SDK might include or depend on a list of known token mint addresses for convenience (for example, knowing the USDC mint on mainnet and devnet). This isn't strictly necessary (the server tells us the mint to pay to), but for developer friendliness, we might allow specifying "USDC" symbol in config and internally map it to the address. This could be powered by Solana token list or a simple mapping. Again, optional sugar.

- **Testing Devnet Environment:** We will need a testing environment. Solana **devnet** is ideal for integration testing – our examples and instructions refer to devnet USDC mint [88]. We'd set up

scenarios: e.g., spin up an Express test server using our server SDK part, then run the client against it to ensure the flow works end-to-end. Additionally, testing on a local validator or test validator could be done for faster feedback. These requirements mean we should also have Solana CLI or tools to airdrop tokens to the payer on devnet for tests.

- **Documentation and Examples:** As part of building the SDK, we should prepare thorough documentation (likely Markdown files or a website) showing how to use it. This includes code examples for both the client side usage in Node and browser, and server side integration. We might base these examples on the minimal code we've examined (simplified for our SDK interface). Given that x402 is new, clear documentation will help drive adoption of our SDK.

To summarize the resource needs: familiarity with Solana JS libraries, understanding of the x402 spec (which we've gained via Coinbase's docs and Solana guides), and references from similar SDKs (Corbits, etc.). By combining these, we can implement an SDK that is robust and aligned with current standards.

Conclusion and Future Considerations

In this deep dive, we outlined how to build an **x402 X Solana SDK** that covers all aspects of the agent payment flow. We chose **TypeScript/Node.js** as the implementation language due to its widespread use among Solana developers [8] and ability to run in both backend and frontend environments [9]. We drew lessons from existing projects like Coinbase's reference (Python) implementation, Corbits (Solana-focused SDK) [17], Mogami (Java SDK) [89], and more, ensuring our design aligns with best practices and covers any gaps. The resulting SDK will enable developers to easily integrate **autonomous micropayments** into their apps – for example, letting an AI agent automatically pay a few fractions of a cent to access an API, leveraging Solana's high-speed, low-cost transactions.

Building this SDK now also positions us to adapt to future developments in the x402 ecosystem. One such development is the introduction of new payment schemes beyond fixed `exact` payments – for instance, a metered **"upto"** scheme (pay up to X based on usage) is contemplated [15]. Our SDK's architecture can be made extensible to support that: e.g., handling streaming or incremental payments, possibly in conjunction with a facilitator. Moreover, as the x402 standard might expand to more blockchains, a future version of our SDK could interface with other networks too. But initially, focusing on Solana gives us a strong foundation, since Solana is currently a premier choice for x402 due to its speed and stablecoin support [5] [6].

In conclusion, an x402 Solana SDK requires bringing together knowledge of HTTP protocols and Solana blockchain programming. By using the prevalent tools (TypeScript and Solana's libraries) and by abstracting the complexity into a developer-friendly package, we empower both AI agents and traditional applications to participate in the **"pay-per-use" web economy** with minimal effort. The end result will be an SDK that truly handles *"most things if not all"* related to x402: from quoting a price, executing a Solana transaction, to verifying and delivering content – thereby accelerating the adoption of internet-native payments in the emerging agentic web.

Sources: The design and assertions above are backed by Solana's official guides and x402 documentation. Solana's guide on getting started with x402 was particularly instructive [90] [91], illustrating the end-to-end flow and listing current SDK efforts. The Solana x402 example code (for a Node client and Express server) provided a concrete blueprint which we've adapted into our plan [92] [46]. Additionally, insights from

industry players – Cloudflare, Coinbase, thirdweb, Mogami – confirm the importance of an easy-to-use SDK for developers and agents [3] [89]. With this solid foundation of research and planning, we are well-prepared to start building the x402 Solana SDK and contribute to the growing **agent economy** on Solana.

---

[1] [2] [5] [6] [13] What is x402? | Payment Protocol for AI Agents on Solana | Solana
https://solana.com/x402/what-is-x402

[3] x402 Protocol Integration Services - Hire Developers to Create AI Agent Payment Systems
https://www.curotec.com/services/technologies/x402-protocol/

[4] [12] [14] [17] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [83] [84] [85] [86] [87] [88] [90] [91] [92] How to get started with x402 on Solana | Solana
https://solana.com/developers/guides/getstarted/intro-to-x402

[7] anchor - why do we still need to write contracts in Rust? - Solana Stack Exchange
https://solana.stackexchange.com/questions/17373/why-do-we-still-need-to-write-contracts-in-rust

[8] [11] JavaScript/TypeScript SDK for Solana | Solana
https://solana.com/docs/clients/official/javascript

[9] How to Start Building with the Solana Web3.js 2.0 SDK - Helius
https://www.helius.dev/blog/how-to-start-building-with-the-solana-web3-js-2-0-sdk

[10] [18] [19] [20] [89] x402 “Builders” List | Who’s Really Powering x402?
https://www.gate.com/learn/articles/x402-builders-list-who-s-really-powering-x402/13436

[15] [16] GitHub - coinbase/x402: A payments protocol for the internet. Built on HTTP.
https://github.com/coinbase/x402

[62] The x402 Protocol: One Line of Code to Unlock Internet-Native ...
https://medium.com/@yhocotw31016/x402-%E5%8D%94%E8%AD%B0-%E4%B8%80%E8%A1%8C%E7%A8%8B%E5%BC%8F%E7%A2%BC%E9%96%8B%E5%95%9F%E7%9A%84%E7%B6%B2%E8%B7%AF%E5%8E%9F%E7%94%9e23aa044a287