ChatGPT

Pay-Per-Crawl MVP Technical Implementation Guide

Overview of the MVP on Base L2

The **Pay-Per-Crawl MVP** is a decentralized protocol that enables content publishers to charge AI crawlers per request (a "pay-per-crawl" model). It is deployed on **Base**, an Ethereum Layer 2 network, to leverage ultra-low transaction costs for micropayments [1]. This MVP focuses on core functionality: enforcing payments for web crawls, processing those payments on-chain in stablecoins, and providing an audit trail. The design is intentionally **modular** and **extensible** – it will serve as the foundation for future features like governance, marketplaces, and advanced pricing models without requiring a complete rewrite.

MVP Components and Roles:
- **On-Chain Smart Contracts (Base L2)**: Handle tokenized licenses, payment processing, and logging of crawl events. (Solidity, deployed to Base)
- **Publisher Edge Worker (Cloudflare)**: Off-chain gateway that intercepts crawler requests and issues payment challenges (HTTP 402) or verifies payment proofs.
- **AI Crawler SDK** (TypeScript and Python): Client libraries that enable AI crawlers to respond to challenges by automatically paying and retrying requests.
- **Publisher Dashboard (Web App)**: A front-end for publishers to onboard, mint their license (NFT), set price terms, and receive the Cloudflare worker script.
- **Account Abstraction (Alchemy Account Kit)**: Infrastructure for ERC-4337 smart accounts and gas sponsorship, allowing crawlers to pay with a smart wallet (no direct ETH needed for gas) [2].

High-Level Flow: A publisher uses the dashboard to mint a **CrawlNFT** (license token) and deploy a Cloudflare worker on their site. When an AI crawler requests a page, the worker responds with "402 Payment Required" and payment details. The crawler's SDK uses an ERC-4337 smart wallet (via Alchemy's Account Kit) to pay the required amount in USDC to the on-chain **PaymentProcessor** contract. The worker verifies the on-chain payment and then allows access to the content, while also logging the event on-chain via the **ProofOfCrawlLedger** for transparency [3] [4]. This end-to-end MVP showcases trustless micropayments for web content, and its architecture is designed such that adding modules like governance or a marketplace later will not require redoing these core components – only extending them.

On-Chain Components (Base L2 Smart Contracts)

All on-chain logic is implemented in **Solidity** and deployed to the Base network. We use a hybrid development stack: **Foundry** for fast Solidity unit testing and **Hardhat** for TypeScript-based scripting, deployment, and integration tests [5] [6]. This "best of both" approach ensures high security and developer velocity. The core contracts in the MVP are kept deliberately simple and non-upgradeable [7] to minimize complexity and attack surface. We will use OpenZeppelin libraries (ERC-721, Ownable, IERC20, etc.) for standardized, secure implementations.

1. CrawlNFT (ERC-721 License Token) – **On-Chain** (Base)
What it is: A non-fungible token representing a content license for a publisher's website. Each publisher who onboards gets one **CrawlNFT** as a credential of their participation in the protocol. It embeds the legal terms of service for crawlers.
Key Features:
- *Inheritance & Access Control*: Inherit from OpenZeppelin ERC721 and Ownable. The contract owner (the Tachi protocol admin for MVP) is the only one allowed to mint new NFTs [8]. This ensures only authorized issuance of licenses.
- *Minting Function*: `mintLicense(address publisher, string calldata termsURI)` mints a new token to the publisher's address. The `termsURI` is an IPFS hash or URI pointing to the license terms (immutable legal agreement) stored off-chain [9]. This URI is recorded in the contract (e.g., in a mapping tokenID -> termsURI or via token metadata). The publisher's wallet will hold the NFT, cementing their ownership of the license.
- *Soulbound (non-transferable)*: Override the `_beforeTokenTransfer` hook to block transfers after the initial mint [10]. This means the NFT cannot be sold or transferred; it represents a specific site's license bound to that publisher. If a transfer is attempted (except minting or perhaps burning in future), the contract will revert the transaction.
- *Extensibility*: In the future, this NFT could be extended with additional metadata or interfaces (e.g., ERC-721 metadata for display, or ERC-5646 for terms licensing). Governance could be introduced to allow updates to terms or revocation. But for MVP, it's a simple, unchanging record of the agreed terms.

Copilot Prompt (CrawlNFT.sol): *"Create a Solidity contract* `CrawlNFT` *inheriting from* `ERC721` *and* `Ownable`*. Include a public function* `mintLicense(address to, string calldata termsURI)` *that can only be called by the contract owner to mint a new token to address* `to`*. Store the* `termsURI` *(e.g., in a mapping) associated with the token ID. Override* `_beforeTokenTransfer` *to prevent any transfers after mint (revert if* `from != address(0)` *and* `to != address(0)`*)."*

2. PaymentProcessor – **On-Chain** (Base)
What it is: A stateless utility contract to accept payments from crawlers and immediately forward them to the publisher. It serves as the on-chain "toll booth" for crawl payments. By using a contract as the payment receiver, we have a consistent on-chain address to monitor for payments (the Cloudflare worker trusts payments only to this contract) and a single point to emit events or handle logic if needed. In MVP it's simple: receive USDC, forward to publisher. In the future, it could take a fee or support complex payment splits, but not yet.

Key Features:
- *USDC Integration*: The contract will interact with a stablecoin (USDC) contract on Base. We assume USDC is an ERC-20 token deployed on Base (or bridged). The PaymentProcessor will use an interface like `IERC20` to transfer tokens.
- *Payment Function*: A function, e.g. `payPublisher(address publisher, uint256 amount)` (or alternatively `pay(uint256 crawlTokenId, uint256 amount)`) will be exposed. The expectation is that the crawler (or rather the crawler's smart wallet) calls this function when a payment is required. - **Process**: The crawler must have approved the PaymentProcessor contract to spend the required USDC amount on its behalf. When `payPublisher` is called, the contract will call `USDC.transferFrom(msg.sender, address(this), amount)` to pull tokens from the crawler's account, then immediately call `USDC.transfer(publisher, amount)` to forward the tokens to the publisher's address. This results in a near-atomic pass-through – the contract should not hold any balance after the call.

- **Validation**: For safety, check that the transfer succeeded (use OpenZeppelin's SafeERC20 or check booleans). Emit an event `Payment(address indexed from, address indexed publisher, uint256 amount)` after forwarding, to log the payment on-chain.
- *Publisher Address Source*: The publisher's address could be provided directly as a function parameter (as above). In an extended design, we might derive it from the CrawlNFT token (e.g., require the caller specify their CrawlNFT token ID, and internally resolve the owner of that token as the payee). For MVP, passing the address explicitly is straightforward.
- *Access Control*: No special access control – any crawler can call `payPublisher` to pay any publisher. The contract doesn't need to restrict who can pay whom (if someone pays the wrong publisher, that's their loss, but not a security issue for the protocol). We do ensure only USDC is handled; if multiple currencies were supported later, we'd have separate functions or an identifier.
- *Extensibility*: Later, this contract could incorporate a fee mechanism (e.g., take a small cut for the protocol treasury before forwarding). It could also support multiple tokens or be replaced by a more advanced marketplace contract. Because the MVP keeps it simple and **non-upgradeable** [7], introducing fees or multi-token support might be done via a new contract version or an upgrade in a v2. The MVP design's simplicity ensures easy replacement if needed.

Copilot Prompt (PaymentProcessor.sol): *"Implement a Solidity contract* `PaymentProcessor` *with a function* `payPublisher(address publisher, uint256 amount) external`*. The function should use* `IERC20` *(for USDC) to transfer* `amount` *from* `msg.sender` *to* `publisher`*. Use* `IERC20.transferFrom` *and ensure it succeeds (consider using SafeERC20). Emit an event* `Payment(address indexed from, address indexed to, uint256 amount)` *when a payment is forwarded. Assume the contract has a reference to the USDC token contract address (e.g., set in constructor)."*

3. ProofOfCrawlLedger – On-Chain (Base)
What it is: A lightweight **append-only log** on the blockchain to record each successful crawl that took place after payment [11]. This provides transparency and auditability – an AI company or regulator can later verify which content was accessed with permission. Each entry in the log ties together the publisher (via their CrawlNFT or address), the crawler, and the content (or at least a content identifier). For MVP, we keep the data minimal.

Key Features:
- *Logging Function*: `logCrawl(uint256 crawlId, address crawler)` or similar. In MVP, we might define `crawlId` as the CrawlNFT token ID for the publisher's site (so it identifies which publisher content was crawled). We could also include a hash or URL of the content, but that might be large or unnecessary on-chain for MVP. A simpler approach: just log the fact that a crawl happened for a given publisher's license. We include the `crawler` address (likely the smart wallet address) to have an identity of who accessed. This function will emit an event, e.g. `CrawlLogged(uint256 indexed crawlTokenId, address indexed crawler)` (and possibly a timestamp or URL if needed).
- *Access Control*: We don't want just anyone spamming this log. In the MVP, the Cloudflare worker will be the one calling `logCrawl` after verifying payment. We can restrict access in a simple way: for example, only the **PaymentProcessor** contract or the CrawlNFT owner (publisher) or the protocol owner can call `logCrawl`. The easiest might be to give the PaymentProcessor or a dedicated signer (like a backend key) the rights. However, since our PaymentProcessor doesn't invoke it directly, we will likely use the protocol's deployer key (same as CrawlNFT owner) to call it via the worker. Thus, we can make the function **onlyOwner** for MVP, meaning only the protocol admin (who also configures the Cloudflare worker with the key) can log

a crawl [12]. This is acceptable in MVP since the scenario is controlled; in a full product, we might have a more complex auth or even remove on-chain restrictions (if we trust that fraudulent logs don't benefit anyone).
- *Storage vs Events*: To keep gas and complexity low, the contract might not store each log in contract storage. Instead, just emitting an event `CrawlLogged` could suffice as the permanent record (since events are on-chain and immutable). We avoid running out of gas by not keeping an ever-growing array. If on-chain querying of logs is needed later, we can introduce an indexing or subgraph.
- *Extensibility*: This ledger could later be expanded to store more data per crawl (content hashes, timestamps, DID of crawler, etc.), or to be permissioned to multiple trusted parties (e.g., both publisher and crawler signatures). However, the existence of a simple on-chain log in MVP already satisfies the core need for transparency. Future compliance tools or analytics dashboards can subscribe to the `CrawlLogged` events to build reports without altering this contract.

Copilot Prompt (ProofOfCrawlLedger.sol): *"Create a Solidity contract* `ProofOfCrawlLedger` *with an owner (Ownable). Include a function* `logCrawl(uint256 crawlTokenId, address crawler) external onlyOwner` *that emits an event* `CrawlLogged(uint256 indexed crawlTokenId, address indexed crawler)`*. This function will be called by an authorized off-chain service (the edge worker) to record that a crawl happened. Keep the contract simple (no need to store data persistently beyond the event)."*

Development & Deployment Notes:
- After writing the contracts, develop **unit tests in Solidity** using Foundry (e.g., test that CrawlNFT minting only works for owner, that PaymentProcessor correctly transfers tokens given a mock ERC20, and that ProofOfCrawlLedger emits events). Use Foundry's fast test runner to fuzz and catch edge cases [6].
- Then write **integration tests in Hardhat (TypeScript)**: for example, simulate a flow where a CrawlNFT is minted, then a PaymentProcessor payment is made by a simulated crawler account (checking that publisher receives funds), and then logCrawl is called. This ensures the contracts work together as expected in a realistic scenario [13].
- Prepare a **deployment script** using Hardhat. Initialize a Hardhat project (TypeScript template via `npx hardhat init`), then integrate it with Foundry (using `hardhat-foundry` plugin or running Forge separately) [5]. The deployment script (e.g., `scripts/deploy.ts`) will: deploy CrawlNFT, deploy PaymentProcessor (pass it the USDC token address on Base testnet/mainnet), and deploy ProofOfCrawlLedger. Then print or save the deployed addresses. Use Alchemy's RPC URL for Base to send the transactions, and verify the contracts on Basescan for transparency [14].
- Deploy to **Base Goerli (testnet)** first for end-to-end testing. After testing, deploy to **Base mainnet** for production MVP.

Copilot Prompt (Deployment Script): *"Using Hardhat and ethers.js, write a script to deploy three contracts: CrawlNFT, PaymentProcessor (with USDC token address), and ProofOfCrawlLedger, in that order. After deployment, log the addresses and save them to a JSON file. Make sure to use the Base network RPC (Alchemy) and handle asynchronous deployment calls. Also include verification steps (using* `hardhat verify`*) for each contract on the explorer."*

Publisher Edge Worker (Cloudflare Gateway)

Where it Lives: Off-chain, at the network edge (Cloudflare Workers). This is a serverless script that runs whenever someone (e.g., an AI crawler) tries to access the publisher's website.

Purpose: The worker acts as the enforcement point for pay-per-crawl. It intercepts HTTP requests **before** they hit the publisher's origin server. If the request is from a known crawler and doesn't include proof of payment, the worker responds with an HTTP **402 Payment Required** status, signaling the crawler to pay. If the request includes a payment proof (in the form of a transaction hash), the worker verifies it on-chain and, if valid, lets the request through to serve the content [15] [4]. Essentially, this is the "paywall" logic implemented in a cloud edge function, ensuring minimal latency to end-users and ease of integration for publishers (just deploy this script on Cloudflare).

Technology: Written in TypeScript (for type safety and alignment with our stack) and deployed via Cloudflare Workers. We will use the **Viem** library (the same underlying library as Wagmi) in the worker because it's lightweight and tree-shakeable, crucial for small edge runtime bundles [16]. (Alternatively, one could use ethers.js or web3, but those are heavier; Viem is chosen to minimize code size and maximize performance).

Key Logic & Steps:
1. **Request Interception & Bot Identification:** The worker's fetch event handler examines each incoming request's headers [17]. It checks the `User-Agent` header against a list of known AI crawler identifiers. For MVP, maintain a static list or regex (e.g., containing "GPTBot", "BingAI", etc.) – this list can be hard-coded or provided by the publisher UI. If the User-Agent matches a target crawler (and possibly if the URL or path is one we want to protect), we proceed to payment enforcement. If not a bot, just `fetch()` to origin normally (bypass the paywall logic for human users and unknown bots in MVP). 2. **Challenge with 402 Payment Required:** If an AI bot is detected and **no valid payment proof** is present, the worker generates a **402 Response** [15]. This response will include custom headers following the draft **HTTP 402 "Payment Required"** standards that some are using for pay-per-call: we'll define headers like:
- `x402-price`: the price in smallest units (e.g., "1000000" for 1 USDC if USDC has 6 decimals, or a human-readable amount like "1.0" – define format clearly). For simplicity, we can use a fixed price (set by publisher during onboarding, e.g. $0.001 per request).
- `x402-currency`: the currency and chain, e.g., "USDC; contract=0x...; chain=Base". Or separate headers for clarity: `x402-currency: USDC`, `x402-network: Base`, `x402-recipient: <PaymentProcessorContractAddress>`. The key is that the crawler knows *where* and *what* to pay [18]. We include the PaymentProcessor contract address (and perhaps the chain ID or name "Base") so the crawler's wallet knows the destination. We don't necessarily include the publisher's address, because PaymentProcessor encapsulates the forwarding logic.
- Optionally, `x402-tokenId` if we want to identify the publisher's CrawlNFT or some license ID. However, since PaymentProcessor doesn't require it explicitly in MVP, we might omit it.
- Also include a standard `Content-Type: application/json` or similar with a body explaining payment required (for debugging or non-automated clients), though crawlers will mainly use the headers.
This 402 response effectively tells the crawler: "Please send X USDC to PaymentProcessor on Base, then retry your request with the transaction hash."
3. **Payment Verification (Authorization Header):** If the request instead contains `Authorization: Bearer <tx_hash>` (meaning the crawler is attempting to prove payment by providing a transaction hash), the worker verifies it [3]. Steps:

- Use the **Viem** client (configured with a Base RPC endpoint, e.g., Alchemy) to get the transaction receipt: `publicClient.getTransactionReceipt({ hash: txHash })` [19].
- Check that the receipt is not null and that `receipt.status` is success. Then inspect `receipt.logs`: specifically, look for a log corresponding to the USDC token's `Transfer` event. We expect to find a `Transfer` from the crawler's address to the PaymentProcessor's address for an amount >= the required price [20]. We can identify the USDC contract's address and parse logs by its ABI or known event signature (`Transfer(address,address,uint256)`).
- Verify **amount** and **recipient**: ensure the amount transferred is **>= price** (allowing a crawler to overpay if they choose, though they normally wouldn't) and that the recipient is exactly the PaymentProcessor contract address we specified. (We might also check that the token contract address matches known USDC to prevent someone sending some other token, but if the logs are fetched specifically from the USDC contract, that's covered).
- (Optional) We could also verify that the PaymentProcessor in turn forwarded the payment via another log (USDC Transfer from PaymentProcessor to publisher). This is an extra check that the publisher indeed got paid. However, since the PaymentProcessor's logic is straightforward, and for MVP we might not complicate by parsing two events, verifying the transfer into PaymentProcessor is sufficient for now. The publisher getting it is guaranteed by contract code.
- If any check fails (no receipt, wrong recipient, insufficient amount, etc.), return a 402 or 403 response indicating payment proof invalid. Otherwise, proceed.

4. **Content Delivery:** If payment is confirmed valid, the worker now fetches the original content from the origin server (i.e., the publisher's actual website) and returns it to the crawler [21]. The `fetch(request)` can be done with the original request or a cloned request (since we might have read the body). Cloudflare allows reading and then re-fetching with modified headers if needed; here we might want to strip the Authorization header when proxying to origin (or the origin could ignore it anyway). The key is to deliver the content that was originally requested.
5. **Asynchronous Crawl Logging:** Before finishing the response, the worker triggers an async task to log the crawl on-chain [21]. Using `event.waitUntil()`, we can perform an asynchronous call to the blockchain without delaying the HTTP response. This involves creating a transaction to call `ProofOfCrawlLedger.logCrawl(crawlTokenId, crawlerAddress)` on Base. We can implement this by using an Ethers/Viem wallet in the worker (Cloudflare Workers can use environment secrets to store a private key, or use something like an API to a secure signer). For MVP simplicity, the private key of the protocol (owner of ProofOfCrawlLedger) can be kept in the worker's secret storage. The worker uses Viem's `walletClient` or a raw fetch to Alchemy's RPC to send the transaction. We include the appropriate data (the CrawlNFT tokenId for the publisher – which we might hardcode in the script or fetch from an env var if one NFT per site). After sending the tx, we typically don't wait for confirmation in the worker (fire-and-forget), but we could log any errors. This call will emit the on-chain `CrawlLogged` event.
6. **Edge Deployment Considerations:** The worker script will be generated or configured by the dashboard with the correct constants (PaymentProcessor address, price, etc.). The publisher will deploy it in their Cloudflare account, associating it with their domain (e.g., via a route like `example.com/*`). Cloudflare's global network ensures the logic runs close to whoever (the crawler) is making requests, reducing latency.

Copilot Prompt (Cloudflare Worker): *"Write a Cloudflare Worker script in TypeScript for a pay-per-crawl gateway. It should listen for FetchEvents. Pseudocode:*

```
if (request.userAgent is AI Crawler && !request.headers.has('Authorization')) {
    return new Response('Payment Required', { status: 402, headers: { 'x402-
```

```
price': '<price>', 'x402-currency': 'USDC', 'x402-recipient':
'<PaymentProcessorAddress>', 'x402-network': 'Base' } });
}
if (request.headers.get('Authorization')) {
    // extract txHash
    // check via JSON-RPC call to Base that PaymentProcessor received payment
(use Viem or fetch RPC)
    if (validPayment) {
       // allow content
       const originResponse = await fetch(request);
       // log crawl asynchronously
       event.waitUntil(sendTxToProofOfCrawlLedger(crawlTokenId,
crawlerAddress));
       return originResponse;
    } else {
       return new Response('Payment verification failed', { status: 402 });
    }
}
return fetch(request); // default for non-bot or if already paid
```

Implement the JSON-RPC call using Viem's `getTransactionReceipt` *and parse logs for USDC Transfer events. Also outline how to use* `waitUntil` *to send a transaction to a contract (ProofOfCrawlLedger)."*

AI Crawler SDK (TypeScript & Python)

Where it Lives: Off-chain, distributed as libraries or modules to be used by AI crawling agents. We provide two language implementations to cater to our target users: one in **TypeScript** (for Node.js environments, JavaScript developers) and one in **Python** (for AI researchers/data scientists who use Python for their scraping and model work). This SDK is essentially the **client** that interacts with the pay-per-crawl protocol from the crawler side.

Purpose: Simplify integration for AI companies. A crawler using this SDK doesn't need to manually implement HTTP 402 handling or blockchain transactions; the SDK abstracts those steps. The end user (crawler developer) can just call something akin to `fetchWithTachi(url)` and get the content, with the payment automatically handled behind the scenes if required.

Core Workflow (What the SDK does): [22] [23]
1. **Initial Request & Challenge Detection:** The SDK provides a function (e.g., `fetchWithTachi(url, options?)`). Internally, this will perform an HTTP GET request to the `url`. This can be done with `fetch` (in Node, via `node-fetch` or built-in fetch if available, or axios/got), or in Python with `requests`.
- If the response is **200 OK** (or any non-402 status), then the content is returned immediately (no payment needed).
- If the response is **402 Payment Required**, the SDK recognizes the pay-per-crawl challenge. It will parse the included `x402-*` headers. It needs to retrieve: the price amount, the currency (which we expect "USDC"), and the recipient address (PaymentProcessor contract on Base) [24]. For example, it may find headers: `x402-price: 1000000`, `x402-currency: USDC`, `x402-recipient: 0x1234...`,

`x402-network: Base`. We parse these values and possibly convert the price to a numeric value (taking into account USDC decimals if needed).
2. **Prepare Payment via ERC-4337 Smart Wallet:** The SDK now automates the payment process using the crawler's **smart contract wallet** (Account Abstraction account): - **Account Abstraction Setup**: The crawler must have an ERC-4337 account ready. In MVP, we assume the crawler developer has already created a smart account (through Alchemy's Account Kit) and has the keys or API to use it. The TypeScript SDK can integrate with Alchemy's AA SDK to manage this. For example, the SDK might include code like:

```
const { PaymentProcessor, USDC } = config; // addresses
const userOp = await smartAccount.execute(
    USDC,
    0,
    USDCInterface.encodeFunctionData("transfer", [PaymentProcessor,
priceAmount]),
    { gasLimit: ..., gasToken: ... }
);
await alchemy.sendUserOperation(userOp);
```

This would construct a UserOperation that calls the USDC contract's `transfer(recipient, amount)` function, sending the required USDC to PaymentProcessor [23]. Alternatively, the UserOperation could call PaymentProcessor's `payPublisher` function directly (which would then internally pull USDC). Either approach works; calling USDC directly is straightforward if the smart wallet holds USDC.
- **Gas Sponsorship**: The smart wallet likely has no ETH, so the Alchemy **Paymaster** will sponsor the gas for this UserOperation (Alchemy's Account Kit handles this behind scenes, possibly charging the account or requiring some stake). The SDK just needs to call the appropriate method to send the UserOperation to the bundler.
- **Sign and Send**: Using the crawler's private key (or a key managed by Account Kit), sign the UserOperation and send it via Alchemy's Bundler RPC. The SDK can use Alchemy's provided methods if using their SDK, or make a direct HTTP POST to the bundler endpoint with the userOp JSON (Alchemy's docs provide an endpoint like `https://base-goerli.g.alchemy.com/v2/<API_KEY>` for bundler).
- **Wait for Confirmation**: The SDK should then wait for the payment transaction to be mined. This can be done by polling the bundler or the RPC for the transaction hash status, or using an event/callback from the AA SDK. In our case, once the bundler accepts the user operation, we will eventually get a transaction hash for the token transfer. The SDK can poll `alchemy.core.getTransactionReceipt(txHash)` until a receipt is available, or use the userOp hash via `eth_getUserOperationReceipt`. For simplicity, polling the tx hash is fine.
3. **Retry the Content Request with Proof:** Once the payment is confirmed on-chain, we have a transaction hash (let's call it `txHash`). The SDK now repeats the original HTTP request to the URL, but includes an **Authorization header** with the token `Bearer <txHash>` [25]. The presence of this header and hash will signal the Cloudflare worker that "here is proof of payment, verify it." The worker will then let us access the content as described earlier.
- The SDK performs this second request. If it returns 200 OK with content, we proceed. (If somehow we got another 402, it means something went wrong in verification, but ideally that should not happen if our payment was correct.)
4. **Return Content**: Finally, the SDK returns the response body (the content) to the caller (the crawler's

code). From the perspective of whoever uses `fetchWithTachi`, it looks like a single call that either returns content or throws an error – all the complex payment handshake is abstracted away.

Additional considerations:
- The SDK should handle errors gracefully: e.g., if the payment transaction fails or times out, it should throw an exception or return an error object. If the content fetch fails, likewise propagate that. Logging can be added for debugging.
- **Python SDK**: The Python implementation will mirror the TS logic. Python doesn't have a native AA SDK from Alchemy at this time (to our knowledge), so we might use the `web3.py` library to construct a transaction, or simply call Alchemy's REST endpoints. One approach: use `requests` to POST a JSON RPC call to Alchemy's bundler. For example, use `eth_sendUserOperation` with the appropriate parameters. Alternatively, rely on Alchemy's Account Abstraction REST if provided. Since this is more complex, we might implement a minimal version: requiring the user to provide an Ethereum private key and using `web3.py` to send a normal transaction (though that defeats the gas sponsorship unless the user's EOA has ETH). A better approach: perhaps use a minimal paymaster that accepts to sponsor if the userOp includes a certain token transfer – Alchemy's paymaster might allow gas fees to be paid in USDC. For MVP, to keep it conceptual, the Python SDK could assume the crawler's wallet is funded with some ETH on Base and just do a direct `USDC.transfer` transaction using web3.py. (This is slightly off the pure AA path but could be a stop-gap for testing in Python if AA in Python is too involved.) In any case, highlight that a Python version would be created for real-world adoption, even if MVP might prioritize the TS version first.

Copilot Prompt (TypeScript SDK): *"Implement an async function* `fetchWithTachi(url: string)` *in TypeScript. It should: - Perform* `fetch(url)`*. - If the response status is 402, parse the* `x402-price`*,* `x402-currency`*,* `x402-recipient` *headers. - Use Alchemy's Account Abstraction SDK to create a UserOperation from the crawler's smart wallet: the operation should transfer the required USDC amount to the given recipient (PaymentProcessor). - Send the UserOperation via Alchemy's bundler and wait for a transaction hash. - Once confirmed, perform a second* `fetch(url)` *with* `Authorization: Bearer <txHash>` *header. - Return the response body. Include error handling for any failed steps."*

Copilot Prompt (Python SDK): *"Implement a Python function* `fetch_with_tachi(url: str)` *that uses the* `requests` *library to GET a URL. If a 402 is returned with x402 headers, use web3.py (or an Alchemy API call) to send a transaction transferring the required USDC to the PaymentProcessor (you can assume you have the private key and web3 instance configured for Base). Wait for the transaction receipt, then send another GET request with an Authorization header containing the tx hash. Return the final content."*

Publisher Dashboard (Onboarding Web App)

Where it Lives: Off-chain, web front-end (React/Next.js application). This is the only user-facing component for the MVP, aimed at content publishers (non-technical users) to guide them through joining the protocol.

Tech Stack:
- **Next.js 14** (with the App Router) for a modern React-based web app [26]. We choose Next.js for its hybrid server-side rendering (for any SEO or static needs) and client-side capabilities for the dApp interactions. Deployment is easy via Vercel, which is ideal for a quick MVP demo.
- **Tailwind CSS** for styling: a utility-first CSS framework that allows rapid UI development without leaving our JSX/TSX files [27]. This will help in quickly customizing the design.

- **Shadcn/UI** component set for pre-built UI components [28]. Rather than using a library like Material UI or Chakra, Shadcn/UI gives us copy-paste, themeable components (built on Radix UI and Tailwind) that we can fully own in our codebase. This aligns with the need for control and customization. We'll use Shadcn for form elements, modals, buttons, etc., ensuring a polished look out of the box.
- **RainbowKit + Wagmi + Viem** for Web3 wallet integration [29]. RainbowKit provides the connect-wallet modal and handles various wallet providers with excellent UX. Wagmi is the React hook library (built on Viem) that simplifies calling smart contracts and reading data from blockchain. Viem under the hood ensures efficient RPC calls and full TypeScript support [30]. Using these, we can easily let the user connect to MetaMask (or Coinbase Wallet, etc.), switch to Base network, and sign transactions to interact with our contracts.

Features & Steps: The Dashboard's primary feature is an **Onboarding Wizard** – a multi-step form that takes the publisher through setup [31] [32]. Each step will be implemented as a React component (we can use a Tabs or Stepper UI from Shadcn/UI for a nice flow). Key steps include:

1. **Connect Wallet:** The first step prompts the user to connect their Ethereum wallet. Using RainbowKit, a "Connect Wallet" button will open the modal. Once connected, ensure the user is on Base network (RainbowKit/Wagmi can prompt network switch). We likely use Base Goerli for testing; RainbowKit supports custom chains so we'll configure Base or Base Goerli in the Wagmi client. This step confirms the user's wallet address (which will be used as the owner of the CrawlNFT).
2. **Site Details & Terms:** Next, the user enters information about their website. For MVP, this might be just the domain name of the site (e.g., "example.com"). We might display the standard **Terms of Service** text that the crawler must agree to (likely a template legal text stored in our app). If we allow customization, the user could edit some fields (like company name) in the terms. Ultimately, these terms (or a reference to them) need to be stored via the CrawlNFT. Strategy: on form submission, upload the terms of service text to IPFS (e.g., using an API like web3.storage or Infura IPFS). This yields a content hash/URI (e.g., `ipfs://Qm...`) which we will pass into the mint transaction as `termsURI`.
3. We can integrate an IPFS upload by including an API route in Next.js or directly calling a pinning service's SDK. For MVP, even embedding a pre-pinned static terms file might be okay to simplify. But dynamic terms per publisher is ideal.
4. Validate inputs (using **react-hook-form** + **zod** for schema validation, as indicated in the plan [33]). E.g., ensure the domain is a valid format and terms are accepted.
5. **Mint CrawlNFT (License):** This is the on-chain step. The user will click "Create License" or similar. Under the hood, we use Wagmi's `useContractWrite` hook (configured with the CrawlNFT contract address and ABI) to call the `mintLicense(publisherAddress, termsURI)` function [34]. Because our contract only allows the owner (the protocol) to mint, we have two possibilities:
6. **Approach A (Simplest for Demo):** Deploy the CrawlNFT contract such that the deployer key (protocol admin) is the same key that runs this front-end (not typical in production, but for demo we could embed the deployer's key in the app – not secure publicly though). This is not ideal.
7. **Approach B:** Instead of calling the contract directly from the user's wallet, call a **backend API** that triggers the mint. For MVP, we could have a backend service (or even a Cloudflare Worker function, or an admin script) listening for a request from the UI to mint. The UI could do something like `await fetch('/api/mint-license', { body: { publisher: address, termsURI } })`, and our server (with the owner key) mints the NFT. The downside is needing backend infra, but this might be acceptable for demo (or we can manually mint and populate the UI for demonstration).

8. **Approach C:** Modify the contract for MVP to allow self-service minting (e.g., remove onlyOwner restriction just for test). The plan text leans on onlyOwner for security [35], so probably they envision the founder running the demo. We'll proceed assuming the founder's wallet (owner) is the one using the app in testing. In any case, the UI should handle the transaction lifecycle (with Wagmi we get `data` and `isLoading` states, etc., to show a spinner and then confirmation).
9. After minting, the UI can display a success message and maybe the token ID. (We could fetch the newly minted token ID by reading events or simply taking the next ID if we know it increments from 1.)
10. **Set Pricing:** The publisher chooses how much to charge per crawl. MVP might have done this in the terms or skip it, but since the Cloudflare worker needs a price, include a step where the user enters a price (in USD or USDC). For example, "$0.005 per request". We'll convert that to a numeric value in USDC's smallest unit (e.g., 0.005 USDC -> 5000 (with 6 decimals)). Keep it simple (no complex tiering, just one price). Possibly default to a recommended price but allow change.
11. This price can be stored off-chain (we will embed it into the Cloudflare script in the next step). Optionally, we could record it in the CrawlNFT's metadata as well for completeness, but on-chain storage of price isn't strictly needed for functionality.
12. **Generate Cloudflare Worker Script:** In the final step, the dashboard presents the user with their customized **Cloudflare Worker code**. This is a text box (or code snippet with copy button) that contains the JS/TS code to deploy. The script will have placeholders replaced with the actual data: the publisher's domain or CrawlNFT ID (if needed), the chosen `PRICE`, the `PAYMENT_PROCESSOR_ADDRESS`, the `USDC_ADDRESS` on Base, and any other config (like network or chain ID). This script is essentially what we described in the Edge Worker section. The user can copy it to clipboard easily.
13. We also provide instructions: e.g., "Go to your Cloudflare dashboard, create a Worker for your site, paste this code, and deploy. Or use Wrangler CLI with `wrangler deploy`." For MVP simplicity, manual deployment via UI is fine [36].
14. **Confirmation & Dashboard:** Once set up, the app might show a "Done!" screen. If we have time, we could also include a simple dashboard view where the publisher can see their license details (e.g., the NFT token ID, the terms link they agreed to, the price they set) and perhaps a placeholder for "Crawl Logs" (which could be fetched by reading events from ProofOfCrawlLedger for their token ID – but that requires indexing; we might skip live data and just note that logs will appear here). This is optional for MVP but demonstrates extensibility (in future, a publisher could log in and see how many times their content was accessed and how much they earned, etc., by reading on-chain data or from a subgraph).

Copilot Prompt (Next.js Onboarding Wizard): *"Using Next.js (React) and Tailwind CSS, create a multi-step onboarding form for the publisher: - Step 1: Connect Wallet (use RainbowKit ConnectButton and show the connected address). - Step 2: Form inputs for Site Domain and an optional custom Terms of Service text area. Provide a default terms text if not customized. - Step 3: Price input (USDC per crawl). - Step 4: A button to mint the license NFT. Use `wagmi`'s `useContractWrite` with the CrawlNFT contract's `mintLicense` function, passing the connected wallet address and an IPFS hash of the terms (you can mock the IPFS part by using a placeholder CID). Handle transaction states (loading, success). - Step 5: On success, display a text area containing a pre-filled Cloudflare Worker script. The script should include the price, PaymentProcessor address, and USDC address. Also show a "Copy to Clipboard" button. Use Shadcn/UI components (like Card, Button, Input) for styling. Validate inputs with Zod (domain format, price > 0). Provide appropriate status messages at each step."*

Dependencies: In implementing the above, ensure to install and configure: `@wagmi/core` and `@rainbow-me/rainbowkit` (plus `viem`), Tailwind (with a configured `tailwind.config.js`), Shadcn

UI (initialize it and import needed components), and any library for IPFS uploads (if used, e.g., `npm i web3.storage`). Use environment variables for things like the Base RPC URL (Alchemy endpoint) and contract addresses.

Account Abstraction with Alchemy's Account Kit (ERC-4337)

To streamline the crawler payment experience, the MVP leverages **Account Abstraction (ERC-4337)** for gasless transactions. Instead of requiring the AI crawler to manage an EOA with ETH for gas, the crawler uses a **smart contract wallet** and gas is sponsored via a **Paymaster**. We utilize **Alchemy's Account Kit** – a hosted ERC-4337 bundler and paymaster service – to avoid the heavy lifting of running our own infrastructure [2]. This integration allows a solo developer to get account abstraction working quickly and reliably.

Setup and Components:
- **Smart Contract Wallet**: Alchemy provides a smart wallet implementation (likely based on standard templates like SimpleAccount). The crawler will have such a wallet. This wallet's address is what we treat as the crawler's identity on-chain (and will be the `msg.sender` for PaymentProcessor, etc.).
- **Bundler Service**: Alchemy's RPC endpoint will accept UserOperation objects. We don't need to run our own bundler; we just send user ops to Alchemy and they handle inclusion on Base.
- **Paymaster**: Alchemy's Account Kit includes a paymaster that can sponsor gas fees. They may allow gas fees to be paid in alternative tokens (in some cases, possibly USDC) or they charge fees to an Alchemy account. For MVP, assume the crawler's operations are sponsored by Alchemy (perhaps with some credits or a simple gas tank mechanism).

How to Integrate in Development:
1. **Obtain API Keys**: Sign up for Alchemy and enable the Account Abstraction (Account Kit) for Base network. Acquire the **bundler URL** and any required keys/tokens. For example, you might get a HTTP RPC endpoint specifically for the bundler (different from the normal JSON-RPC endpoint). [37]
2. **Install Alchemy's AA SDK**: In the TypeScript environment (crawler SDK or even in the front-end if needed), install the SDK (e.g., `@alchemy/aa-sdk` or the relevant package). Alchemy's blog/docs (e.g., *Introducing Account Kit* [38]) provide code examples. Typically, you'd do something like:

```
import { Alchemy, UserOperationBuilder, getAccount } from "@alchemy/aa-sdk";
const alchemy = new Alchemy({ apiKey: ALCHEMY_API_KEY, network: "base-
goerli" });
const smartAccount = await getAccount({ /* config for owner private key,
paymaster etc. */ });
```

(Pseudo-code – the actual API may differ.) This will handle the creation or retrieval of a smart account for a given owner.
3. **Create/Configure Smart Wallet**: If the crawler doesn't have a deployed smart wallet, use the SDK to deploy one (most AA SDKs will deploy the account contract on first use if needed). For testing, one could also manually deploy an account contract using a factory, but SDK likely simplifies it. Ensure the wallet has USDC funds (you can fund it by transferring USDC from a normal wallet to the smart wallet address).
4. **Building User Operations**: Use Alchemy's utilities or manual construction to create the UserOp for the

payment: - Set the sender to the smart wallet address. - For the action, encode the call to either the USDC contract or PaymentProcessor with the correct parameters for the payment. - Alchemy's SDK might have a `smartAccount.sendTransaction({to, data, value})` that internally creates the user op with proper nonce, gas, etc., and sends it. If not, use `UserOperationBuilder`. The SDK will fill in things like `maxPriorityFee`, `maxFee`, and attach the paymaster info if using their paymaster.
- The paymaster info might require an API call to get a paymaster signature or something. Some kits abstract that away.
5. **Sending the UserOp**: Call the bundler endpoint via the SDK (e.g., `alchemy.sendUserOperation(userOp)`). This returns a userOp hash or directly the transaction hash upon mining. The flow from there we covered: wait for confirmation and then proceed.
6. **Testing**: Test the AA flow on Base Goerli. For example, use the crawler SDK to pay a small amount and ensure the bundler processes it. Check on Base Goerli explorer that a **UserOperation** was included and a **token transfer** occurred. Also test scenario where the smart wallet has no ETH – it should still work if paymaster sponsorship is correct (the bundler will typically bundle a paymaster call that pays the gas in something).

By using Account Kit, the developer avoids deploying a custom bundler or writing a paymaster contract, which would be infeasible in a quick MVP. It accelerates time-to-market [2]. Alchemy's service is production-ready and will handle scale as we go from test to mainnet.

Copilot Prompt (Account Kit Integration Example): *"Show how to integrate Alchemy's Account Kit in a Node.js script. For example, create a smart account for a given owner private key and craft a UserOperation that calls an ERC-20 token contract. Use Alchemy's SDK methods to send this UserOperation on the Base network and wait for the transaction to be mined. Provide code for initializing the Alchemy AA provider and sending a simple token transfer via account abstraction."*

(The actual output will depend on Alchemy's API; we expect something like setting up an `EthersProviderAdapter` *with Alchemy's RPC, then using* `SimpleAccount` *class to send transactions.)*

Integration & Deployment Steps

With all components built, here's how to integrate and deploy the MVP system step-by-step:

1. **Smart Contracts Deployment:** Deploy the CrawlNFT, PaymentProcessor, and ProofOfCrawlLedger contracts to **Base Goerli** (for testing). Use the Hardhat deploy script created earlier. Confirm their addresses and store them in the front-end configuration [39]. Once tested, deploy to **Base Mainnet** for the live MVP. Make sure to verify contracts on Basescan and possibly write down the ABI and addresses for use in the front-end and worker.
2. **Configure the Front-End:** In the Next.js dashboard app, update environment variables (e.g., in `.env.local`):
3. `NEXT_PUBLIC_BASE_RPC_URL` (Alchemy/Infura RPC for Base),
4. `NEXT_PUBLIC_CRAWLNFT_ADDRESS`, `NEXT_PUBLIC_PAYMENTPROCESSOR_ADDRESS`, `NEXT_PUBLIC_PROOF_LEDGER_ADDRESS`,
5. `NEXT_PUBLIC_USDC_ADDRESS` (the USDC token on Base),
6. possibly `NEXT_PUBLIC_ALCHEMY_API_KEY` for using it in client (though we might keep Alchemy usage server-side or in SDK).

Ensure wagmi is configured with the Base chain and these addresses. Test the Next.js app locally (with `pnpm run dev` or similar) connected to Base Goerli: go through the onboarding flow and see that transactions pop up in your wallet. Deploy the front-end to **Vercel** or a static host when ready [40].

7. **Publisher Onboarding (Test Run):** Using the deployed front-end on Base Goerli, simulate a publisher onboarding: connect a wallet (one that is the deployer if using onlyOwner logic, or adjust as needed), input a dummy site (you can even set up a test site), mint the NFT, set a price, and copy the Cloudflare worker code. Verify on Base Goerli explorer that the CrawlNFT was minted (and terms URI stored correctly).
8. **Cloudflare Worker Deployment:** Go to the Cloudflare dashboard (or use `wrangler CLI`). Create a new Worker script. Paste the code from the dashboard. Set the route to match your test site domain (for a quick test, you could use a Cloudflare Pages site or any domain you control). If using `wrangler`, you might do `wrangler secret put` to add the private key for logging (if the worker needs it), then `wrangler publish`. After deployment, the Worker should be active, intercepting requests to that domain [36].
9. **Crawler Setup:** Configure the crawler environment for testing:
10. If using the TypeScript SDK, write a small Node.js script or test that calls `fetchWithTachi("https://your-test-site/page")`. Before running, ensure the crawler's smart wallet is set up: e.g., use Alchemy's tools or the SDK itself to create the account. Fund the smart wallet with a few USDC tokens on Base Goerli (you can get some from a faucet or from your main wallet). If Alchemy's paymaster requires any specific setup (like whitelisting the token or pre-funding the paymaster), do that via their dashboard.
11. If testing manually, you could also simulate a crawler by just doing a cURL or browser fetch to the URL and manually sending a transaction, but using the SDK is closer to real usage.
12. **End-to-End Test:** Trigger the test crawl:
13. Initially, the crawler SDK should get a 402 response. Check that the 402 and headers are received correctly. The SDK then sends the UserOperation to pay. Monitor the console or Alchemy dashboard: you should see a transaction for USDC transfer from the smart wallet. Once mined, the SDK should retry the request with the tx hash. This time the worker should validate and allow the content. The content (e.g., HTML of the page) is returned to the SDK. Confirm that the content is indeed fetched.
14. Now verify on-chain logs: Check the PaymentProcessor's balance (should be zero, but you can see the USDC transfer events to it and out to publisher in Base Goerli's explorer logs). Check the ProofOfCrawlLedger contract events – you should see a `CrawlLogged` event for the publisher's token ID [20] [41]. This confirms the final step.
15. **Deployment to Production:** After successful testing on Goerli, repeat the process on Base mainnet: deploy contracts (with perhaps an updated terms URI if needed), update the front-end config to mainnet addresses, and ask a friendly publisher to try it out with real USDC (maybe using small amounts for safety). Alchemy's Account Kit on mainnet would handle real transactions similarly (make sure the paymaster is enabled for mainnet on your account, and that the crawler's smart wallet has USDC).

Throughout these steps, use VS Code and GitHub Copilot to accelerate development. For example, when writing the Cloudflare worker, start by describing the logic in comments and let Copilot suggest code structure. Similarly, for each contract and front-end component, leverage prompts (like those given above) to have AI generate boilerplate, which you then refine.

Extensibility and Future Directions

One of the goals in designing this MVP is to ensure the architecture can be extended into a full-fledged protocol (with governance, a marketplace, etc.) without scrapping what's been built. Here's how our design supports that:

- **Modular Smart Contracts:** The separation of concerns is clear (NFT for identity/legal terms, PaymentProcessor for payment routing, Ledger for logging). This means each piece can be upgraded or replaced independently. For instance, introducing a marketplace might involve deploying a new contract that perhaps interacts with the CrawlNFTs (e.g., a registry or exchange), but the existing NFTs remain valid credentials. Governance could be layered on by giving a DAO control over certain parameters (like global fees or upgrades) via the Ownable/onlyOwner mechanism we already use – for example, the DAO's multi-sig could become the owner of CrawlNFT contract to allow or pause minting, etc., without changing the NFT logic itself. The MVP's use of standard tokens and contracts makes future integration straightforward.
- **Scalability of Base & Multichain:** We chose Base for cost and speed; if in the future the protocol needs to be multichain (other L2s or L1s), we can redeploy the contracts on those chains. Because we rely on stablecoin (USDC) and not chain-specific currency, as long as USDC (or a bridged equivalent) is on another network, the PaymentProcessor concept works the same. The Cloudflare worker could be configured per publisher to point to different chain endpoints if needed. The Account Abstraction approach is also replicable on any chain that supports ERC-4337 (many will).
- **Adding a Marketplace:** Currently, crawlers discover a publisher's terms and price only by getting a 402 response from the site. A marketplace service can be built on top to make this discovery easier (crawlers could query a catalog of participating sites and prices). To implement this, one could use an off-chain index (subgraph) of all CrawlNFTs and perhaps a new contract where publishers register their NFT and price. Because our NFTs and logs are on-chain, much of the data needed for a marketplace (like who's participating, how often transactions happen) is already accessible [42]. We would **not** need to redesign the payment flow – just provide a new interface for negotiation. This demonstrates the benefit of having the NFT as a stable anchor representing the license.
- **Governance & Tokenization:** We might introduce a governance token to decentralize control. For example, deciding acceptable terms or setting default prices could be governed. We can do this by upgrading the ownership of contracts to a DAO contract (like OpenZeppelin's Governor or a multi-sig). Because MVP contracts are simple, handing over control is easy. Additionally, a reward mechanism (maybe a token reward for popular content or for crawlers that behave well) could be layered by integrating the ProofOfCrawlLedger events with a token distribution contract. None of these enhancements break the existing MVP flow; they **add** to it.
- **Security Enhancements:** As more value flows through the system, we'd implement things like upgradable contracts or more complex payment verification (to prevent edge cases). The MVP's simplicity (no upgradeability, single-owner control) is okay short-term [7], but long-term, a proxy pattern might be introduced for PaymentProcessor to upgrade logic (e.g., allow fee take). We ensured MVP logic is simple enough that replacing a contract later (with a migration script) is feasible if needed (for instance, deploying `PaymentProcessorV2` and updating the worker scripts to point to the new address).
- **Logging and Analytics:** The on-chain ledger ensures any third party can build analytics. As usage grows, a subgraph (The Graph) could index `CrawlLogged` events to provide real-time insights to publishers (number of crawls, revenue earned) and to crawlers (cost spent per site). The MVP's

logging contract is designed to be easy to index (flat event with key IDs). We wouldn't need to refactor to get this data; just add an indexing layer.

- **Support for Multiple Content Types or Hierarchy:** Currently one NFT = one site. In future, publishers might want multiple NFTs for different sections of content or different licenses. We can allow the CrawlNFT contract to mint multiple tokens per publisher (it already can, since tokenId just increments). The worker could potentially enforce different prices for different content categories by referencing different CrawlNFT IDs (this would require passing some identifier in the 402 headers). These are extensions possible with minor tweaks, not redesigns.

In summary, the MVP implementation outlined above provides a clear **step-by-step guide** to building a Pay-Per-Crawl system that is functional yet simple. Each component was chosen for quick implementation by a solo developer (with Copilot assistance) and for providing a path to evolve into the comprehensive Tachi protocol described in the research [43] [44]. By following this guide and using the prompts, one can assemble the MVP and be confident that it won't be throw-away work – it's the scaffold upon which the full decentralized pay-per-crawl ecosystem can be built.

---

[1] Pay-Per-Crawl Protocol Viability Analysis .docx
file://file-HxeHYTdXZ7rVxxbQeBypHd

[2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] Tachi: Pay-Per-Crawl Technical Plan .docx
file://file-BTEdg31L3q6qeM3FgnL6P9

[16] Pay-Per-Crawl Startup Technical Plan .docx
file://file-S2dRikhHNvZGa4HdReh9wz