Technical Plan: Architecting and Building a Decentralized Pay-Per-Crawl Protocol

Section 1: Foundational Technical Strategy & Stack Selection

This section establishes the core architectural pillars of the Pay-Per-Crawl protocol. It addresses the most critical, high-level technology decisions that will influence the entire development lifecycle, justifying each choice based on rigorous analysis of technical trade-offs, market conditions, and the primary goal of achieving a scalable, secure, and economically viable system.

1.1 Blockchain and L2 Selection: Formalizing the Choice of Base

The fundamental economic premise of the Pay-Per-Crawl protocol is the viability of high-frequency, low-value micropayments at a global scale. The viability analysis concludes that this model is not only feasible but represents a core competitive advantage over incumbent, fiat-based systems.[1] This requirement immediately disqualifies Ethereum Layer 1 (L1) due to its high and volatile gas fees, making a Layer 2 (L2) scaling solution a mandatory architectural component.

After evaluating the landscape of available L2s, **Base** is selected as the foundational blockchain for the protocol's initial deployment. This decision is predicated on a confluence of technical, economic, and strategic factors:

- **Economic Feasibility:** The protocol's success is directly contingent on transaction costs being several orders of magnitude lower than the payment itself. Base, as an Optimistic Rollup, offers extremely low transaction fees. Current estimates place the cost of a token transfer via an ERC-4337 smart contract account on Base at less than $0.0001.[1] This ultra-low cost structure is the bedrock of the protocol's economic model, enabling true per-request micropayments that are impossible on fiat rails or Ethereum L1.[1]
- **EVM Compatibility and Tooling:** Base is fully Ethereum Virtual Machine (EVM) compatible.[3] This is a critical factor for de-risking development and accelerating time-to-market. It grants the project immediate access to the entire, mature ecosystem of Ethereum development tools, including frameworks like Foundry and Hardhat, smart

contract libraries like OpenZeppelin, and the vast pool of developer talent proficient in Solidity.[4] This avoids the need to invest in bespoke tooling or specialized expertise required by non-EVM chains.

- **Ecosystem and Liquidity:** As an L2 incubated by Coinbase, Base benefits from unparalleled ecosystem support, deep liquidity for key stablecoins like USDC, and robust, trusted on/off-ramps.[3] This is not a trivial consideration; it directly addresses the practical business needs of both publishers and AI companies. Publishers need a simple way to convert their USDC earnings into fiat, and AI companies need a reliable channel to fund their crawler wallets. The deep integration with the Coinbase ecosystem streamlines this entire process.[8]
- **Security and Decentralization Model:** Base leverages the security of the Ethereum mainnet for transaction finality and data availability.[3] By posting its transaction data to Ethereum L1, Base inherits the security and decentralization guarantees of the most battle-tested smart contract platform, providing a trusted and credibly neutral foundation for the protocol's on-chain audit ledger.[1]

1.2 Smart Contract Development Environment: Foundry vs. Hardhat

The choice of development framework is a critical decision that profoundly impacts developer workflow, testing philosophy, team velocity, and ultimately, the security of the final product. The two leading frameworks in the EVM ecosystem are Hardhat and Foundry.[6] While either could be used, a careful analysis of their respective strengths and weaknesses suggests a hybrid approach is optimal for this project.

Foundry Analysis:

Foundry is a Rust-based toolkit known for its exceptional performance, with significantly faster compilation and test execution times compared to its JavaScript-based counterparts.10 Its most compelling feature is that it allows tests to be written directly in Solidity. This reduces the cognitive overhead of context switching between languages and encourages developers to achieve deeper mastery of Solidity itself.10 Furthermore, Foundry comes with powerful, natively integrated tools that are invaluable for security-focused development, including robust fuzz testing to discover edge-case vulnerabilities and detailed call tracing for debugging complex interactions.10

Hardhat Analysis:

Hardhat remains the most widely adopted Ethereum development framework, boasting a vast and mature plugin ecosystem that can accelerate development for a wide range of tasks.6 Its tests and scripts are written in JavaScript or TypeScript, which is advantageous for teams with strong web development backgrounds and allows for the seamless reuse of code and libraries between on-chain and off-chain components.12 Hardhat's local network simulation and debugging capabilities, including detailed stack traces, are highly refined.13

Recommendation: A Hybrid Approach

The recommended strategy is to leverage the strengths of both frameworks. Foundry will be

the primary tool for core smart contract development and unit testing. Its speed and security-centric features, particularly fuzzing and Solidity-native tests, are best suited for building and rigorously validating the on-chain logic where security is paramount. Hardhat will be used for more complex integration testing, deployment scripting, and managing interactions with off-chain systems. Its powerful JavaScript/TypeScript environment is ideal for scripting deployments to various networks (testnet, mainnet), running integration tests that simulate the full end-to-end flow (including the edge worker and crawler SDK), and managing contract upgrades. This hybrid model provides a "best of both worlds" development environment, optimizing for both security and operational efficiency.13

1.3 Core Client-Side & Off-Chain Libraries

To ensure a cohesive and performant system, a standardized set of libraries for off-chain components is essential.

- **Blockchain Interaction: Viem** is selected as the primary TypeScript library for all client-side and server-side interactions with the Base blockchain. Viem is a modern, lightweight, and highly performant alternative to the more traditional ethers.js library.[14] It offers superior tree-shakability, which is important for keeping the edge worker's bundle size small, and its rigorous TypeScript support provides enhanced type safety. As the underlying library for popular tools like Wagmi, it represents the modern standard for EVM interaction.[15]
- **Publisher Gateway (Edge Worker):** The gateway will be developed in **TypeScript**. This choice ensures type safety, which is critical for a component handling financial logic, and aligns with the modern web development ecosystem. The primary deployment targets are Cloudflare Workers and Vercel Edge Functions, as their widespread adoption and simple deployment models directly address the low-friction integration requirement for publishers.[1]
- **AI Crawler SDK:** To maximize adoption, the SDK will be provided in two languages: **TypeScript/JavaScript (for Node.js)** and **Python**. While TypeScript aligns with the rest of the off-chain stack, Python is the lingua franca of the AI/ML community. Providing a native Python SDK is non-negotiable for achieving adoption among the target user base of ML engineers and data scientists.

1.4 Account Abstraction (ERC-4337) Infrastructure Strategy

The protocol's architecture relies heavily on ERC-4337 Account Abstraction to enable autonomous, gas-sponsored operation for the AI crawlers.[1] This standard introduces new, specialized infrastructure components: the
Bundler, which packages and relays UserOperation objects to the blockchain, and the **Paymaster**, a smart contract that can sponsor gas fees on behalf of users.[18] The project

faces a critical "build vs. buy" decision for this infrastructure.

While self-hosting these components offers maximum control and avoids platform dependency, it represents a significant engineering and operational undertaking. Running a production-grade Bundler requires deep expertise to ensure high availability, low latency, and resilience against Denial-of-Service (DoS) and other attacks.[20]

Given that the primary risks identified in the viability analysis are related to go-to-market execution and legal novelty, not a lack of technical features, the most prudent strategy is to offload this infrastructural complexity.[1] Therefore, for the MVP and initial scaling phases, the project will leverage a third-party Infrastructure-as-a-Service (IaaS) provider.

Recommendation: Alchemy Account Kit

Alchemy's Account Kit is selected as the IaaS provider for ERC-4337 infrastructure.22 It offers a complete, vertically integrated toolkit that includes a production-ready Bundler API, a Gas Manager API for programmable gas sponsorship, and a secure, gas-optimized Light Account smart contract implementation.[22] By using Account Kit, the development team can abstract away the immense complexity of running this infrastructure, allowing them to focus their resources on building the core protocol logic and accelerating the time-to-market for the MVP. This strategic trade-off—sacrificing some measure of decentralization for a massive gain in development speed and reliability—directly addresses the most pressing business challenges of the startup. The modularity of the aa-sdk also provides flexibility, allowing for the integration of various signer types, which will be crucial for the crawler's key management.[24]

The following table summarizes the foundational technology choices that will guide the development of the Pay-Per-Crawl protocol.

Table 1: Technology Stack Selection for MVP

Category	Selection	Justification	Key Research Snippets
Blockchain	Base (Ethereum L2)	Ultra-low transaction fees, EVM compatibility, strong ecosystem support, USDC liquidity.	[1]
Smart Contract Language	Solidity	Most mature language, largest developer community, extensive tooling and libraries.	[5]
Development Framework	Foundry (for unit tests) & Hardhat (for integration/deployment)	Combines Foundry's speed and security features with Hardhat's scripting power and ecosystem.	[6]
Core On-chain Libraries	OpenZeppelin Contracts	Industry standard for secure, audited,	[6]

		reusable contract components (ERC-721, Access Control).	
Client-Side Library	Viem	Modern, lightweight, type-safe, and performant alternative to ethers.js.	[14]
Edge Worker Language	TypeScript	Provides type safety and aligns with modern web development standards.	[16]
AA Infrastructure	Alchemy Account Kit (Bundler & Gas Manager)	Production-ready, vertically integrated IaaS to accelerate time-to-market and reduce operational complexity.	[22]

Section 2: The Minimum Viable Product (MVP) - Architecture & Build Plan

This section defines the precise scope, architecture, and end-to-end functionality of the Minimum Viable Product (MVP). The MVP is laser-focused on the most direct path to validating the core hypothesis of the business: enabling a publisher to receive an on-chain micropayment from an AI crawler in exchange for access to a single piece of content.

2.1 MVP Guiding Principles

The development of the MVP will be governed by a set of strict principles designed to maximize learning and minimize time-to-market.

- **Principle 1: Solve the Core Problem First.** The singular goal of the MVP is to successfully execute one atomic, end-to-end pay-per-crawl transaction. All other features, such as advanced pricing, governance, or secondary markets, are explicitly deferred.
- **Principle 2: Minimize Publisher Friction.** As identified in the viability analysis, publisher adoption is contingent on a near-frictionless onboarding experience.[1] The process for a publisher to begin monetizing their content must be as simple as connecting a wallet, filling out a form, and deploying a pre-configured software component to their existing web infrastructure.

- **Principle 3: Prioritize AI Crawler Developer Experience.** The primary user of the AI Crawler SDK is an ML engineer, not necessarily a Web3 expert. The SDK must abstract away the underlying blockchain complexity, particularly the construction of ERC-4337 UserOperations, to provide a simple and intuitive API.[27]
- **Principle 4: Defer Decentralization for Speed.** To achieve the necessary velocity for the MVP, elements that contribute to full decentralization but are not essential for validating the core economic loop will be deferred. This includes on-chain governance, secondary token markets, and the operation of self-hosted infrastructure. The initial focus is on validating the model within a more controlled environment.

2.2 MVP System Architecture

The MVP architecture consists of three primary actors—the Publisher, the AI Crawler, and the Blockchain—and a set of interacting software components.

- **Actors:**
  - **AI Crawler:** An autonomous agent whose goal is to fetch web content. It is equipped with the Pay-Per-Crawl SDK.
  - **Publisher:** A content creator who wants to monetize access to their website's data.
  - **Blockchain (Base L2):** The shared, immutable ledger that facilitates payment and provides a transparent audit trail.
- **Components:**
  - **On the AI Crawler side:**
    - **Crawler Logic:** The core application logic that determines which URLs to fetch.
    - **Pay-Per-Crawl SDK:** A library that handles the entire 402 payment and verification flow.
    - **ERC-4337 Smart Wallet:** The crawler's on-chain identity and treasury, controlled by the SDK.
  - **On the Publisher side:**
    - **Publisher Web App:** A simple web interface for onboarding, configuration, and viewing earnings.
    - **Publisher Gateway (Edge Worker):** A piece of serverless code deployed on the publisher's CDN (e.g., Cloudflare) that acts as the protocol's enforcement point.
  - **On-chain (Base):**
    - **Crawl-NFT Contract:** An ERC-721 contract representing the publisher's license.
    - **PaymentProcessor Contract:** A simple contract that receives and forwards USDC payments.
    - **Proof-of-Crawl Ledger Contract:** A contract that logs all verified crawl events.

- **Third-Party Services:**
  - **Alchemy Account Kit:** Provides the Bundler and Paymaster API endpoints required for ERC-4337 functionality.
  - **RPC Provider:** A node service (e.g., Alchemy, QuickNode) used by the Publisher Gateway to verify transactions.

2.3 Scoped Feature Set: In vs. Out

To maintain focus, the scope of the MVP is tightly constrained.

IN SCOPE for MVP:

- A simple web application for publishers to:
  - Connect a Web3 wallet (e.g., using RainbowKit or Web3Modal).[29]
  - Mint their site's unique Crawl-NFT (ERC-721).
  - Set a single, site-wide price-per-crawl in USDC.
  - Download a pre-configured edge worker script for one target platform (e.g., Cloudflare).
  - View a basic dashboard of their total earnings and a list of recent crawl events, populated by reading events from the Proof-of-Crawl contract.
- The Publisher Gateway edge worker with logic for issuing 402 challenges and verifying payment proofs.
- The three core smart contracts as specified in Section 3: CrawlNFT, PaymentProcessor, and ProofOfCrawlLedger.
- The AI Crawler SDK provided as a TypeScript/Node.js package.
- Full integration with a third-party ERC-4337 Bundler and Paymaster service (Alchemy).[22]

OUT OF SCOPE for MVP:

- The advanced rights model using fungible ERC-20 CrawlTokens and secondary market trading.
- The Protocol DAO, governance token, and any on-chain fee-splitting mechanisms.
- Granular pricing controls (e.g., different prices for different pages or content types).
- Support for any blockchain other than Base or any stablecoin other than USDC.
- Self-hosted Bundler or Paymaster infrastructure.
- Advanced analytics, reputation systems, or fraud detection beyond the basic on-chain verification.

2.4 End-to-End MVP Transaction Flow (The "Happy Path")

This detailed flow describes the complete, successful lifecycle of a single pay-per-crawl interaction, creating a tangible example of the protocol in action. This sequence is not just a technical process; it is structured to form a legally coherent electronic contract.

1. **Setup (One-time):**

    - A publisher uses the protocol's web application to connect their wallet and mint a unique Crawl-NFT. The NFT's metadata permanently links to the publisher's Terms of Service, establishing the **Offer**.[1]
    - The publisher deploys the provided edge worker to their website's infrastructure. The worker is configured with the publisher's wallet address and the addresses of the protocol's smart contracts.
    - An AI company integrates the SDK into its crawler application. The company provisions an ERC-4337 smart wallet for its crawler, funds it with USDC on the Base network, and configures a gas sponsorship policy via Alchemy's Gas Manager API. This allows the crawler to operate autonomously without needing a native gas token balance.[18]
2. **Crawl Request:** The AI Crawler makes a standard HTTP GET request to a target URL on the publisher's website.
3. **Payment Challenge:** The Publisher Gateway (edge worker) intercepts the request. It identifies the request as coming from a known AI crawler (e.g., via User-Agent or IP address in the MVP) and sees that no valid payment proof is included. It constructs and returns an **HTTP 402 Payment Required** response.[1] Following the x402 standard, the response headers contain the necessary payment information: the price, currency (USDC), L2 network (Base), and the recipient address (the PaymentProcessor smart contract).[1]
4. **Payment Construction:** The AI Crawler's SDK receives and parses the 402 response. It uses this information to build an ERC-4337 UserOperation object.[27] The callData field of this operation is meticulously crafted to encode a call to the crawler's smart wallet, instructing it to execute a transfer of the specified amount of USDC to the PaymentProcessor contract.[33]
5. **Payment Execution:** The SDK sends the signed UserOperation to Alchemy's Bundler API endpoint.[22] The Bundler validates the operation, packages it into a standard Ethereum transaction, and submits it to the EntryPoint contract on the Base network.[18] The EntryPoint contract orchestrates the execution: it verifies the crawler's signature and, if a Paymaster is used, ensures the Paymaster pays the gas fee before executing the USDC transfer from the crawler's wallet to the PaymentProcessor.[20] This on-chain payment constitutes the crawler's explicit **Acceptance** of the publisher's offer.
6. **Content Request with Proof:** The SDK continuously polls a Base RPC node for the transaction receipt. Once the transaction is successfully mined and confirmed, the crawler makes a second HTTP GET request to the original URL. This time, it includes an Authorization header containing the confirmed transaction hash as a cryptographic proof of payment.
7. **Payment Verification & Content Delivery:** The Publisher Gateway intercepts the second request. It extracts the transaction hash from the header and uses a Viem client to query a Base RPC node for the transaction receipt.[34] The gateway performs a series

of critical checks:
    - The transaction was successful (status is 'success').
    - The recipient of the USDC transfer (decoded from the event logs) was the correct PaymentProcessor contract.
    - The amount of USDC transferred matches the required price.
    - The transaction originated from a wallet associated with the crawler.
      Upon successful verification, the gateway fetches the requested content from the origin server and returns it in the HTTP response. This delivery of content represents the Consideration for the payment, completing the contractual exchange.
8. **Audit Log:** After successfully serving the content, the gateway asynchronously triggers a final on-chain transaction. It calls the logCrawl function on the Proof-of-Crawl ledger contract, immutably recording the crawler's identity, a hash of the content URL, and a timestamp.[1] This creates a permanent, publicly auditable record of the entire interaction.

Section 3: Core Component Deep Dive: Smart Contract Engineering (MVP)

This section provides the detailed technical specifications for the on-chain components of the MVP. The smart contracts are designed for simplicity, security, and low gas cost, prioritizing the validation of the core protocol loop over feature completeness. The contracts will be written in Solidity and developed using the recommended hybrid Foundry/Hardhat environment. The deliberate choice to make these initial contracts simple and non-upgradable is a strategic one, aimed at minimizing security risks and accelerating the audit process. The complexity inherent in upgradable proxy patterns and sophisticated on-chain logic introduces a significant attack surface, which is an unnecessary risk for an MVP whose primary goal is to validate market demand and legal hypotheses.[32]

3.1 CrawlNFT.sol (ERC-721)

This contract serves as the immutable, on-chain anchor for a publisher's participation in the protocol and their governing terms of service.

- **Standard:** The contract will inherit from OpenZeppelin's battle-tested ERC721.sol for the non-fungible token standard and Ownable.sol for simple access control.[6]
- **Purpose:** Each participating publisher website will be represented by a single, unique Crawl-NFT. This token is not meant for trading but acts as a verifiable, on-chain credential and a pointer to the legal framework governing the content access agreement.

- **State Variables:**
    - string private _termsOfServiceURI: This variable stores a persistent link to the master Terms of Service document. To ensure immutability and decentralization, this will be an IPFS hash (e.g., ipfs://...). It is set once at the time of minting.
- **Core Functions:**
    - constructor(address initialOwner, string memory termsURI): The constructor is called when a publisher onboards via the web app. It mints the single NFT (token ID 1) directly to the initialOwner's wallet and permanently sets the _termsOfServiceURI.
    - getTermsURI() public view returns (string memory): A public, gas-free view function that allows anyone (including crawlers and legal observers) to retrieve the URI for the terms of service, creating a transparent link between the on-chain asset and its governing legal text.
- **Security Considerations:** To reinforce its role as a non-tradable site credential, the contract will override the _beforeTokenTransfer hook from ERC721.sol to revert on any transfer attempts after the initial minting. This prevents the license anchor from being separated from the operational control of the website.

3.2 PaymentProcessor.sol

This is the financial heart of the MVP, designed to be a simple and secure conduit for funds.

- **Purpose:** The PaymentProcessor is a stateless contract whose sole responsibility is to receive USDC payments from AI crawlers and immediately forward them to the appropriate publisher. It is the designated recipient in the 402 response and the target of the crawler's payment UserOperation.
- **State Variables:**
    - address public immutable usdcTokenAddress: The contract address of the canonical USDC token on the Base network. Set in the constructor to prevent modification.
    - address public immutable protocolTreasury: The address designated for collecting protocol fees. For the MVP, this will be set to a placeholder address (e.g., the zero address), and the fee will be 0%.
- **Core Functions:**
    - receivePayment(address publisherWallet, uint256 amount): This is the primary entry point for payments.
        - **Mechanism:** It is designed to be called through the USDC contract's transfer function (or a similar pattern like transferAndCall if available). The msg.sender will be the crawler's smart wallet, which is authorized by the EntryPoint contract.
        - **Logic:** The function pulls the specified amount of USDC from the msg.sender. It then immediately executes a transfer of the entire amount to the publisherWallet address provided as an argument.

        - **Event Emission:** Upon successful transfer, it emits a PaymentReceived event, logging the publisherWallet, the crawlerWallet (msg.sender), and the amount paid. This event provides an easily indexable on-chain record of financial transactions.

3.3 ProofOfCrawlLedger.sol

This contract creates the immutable, transparent, and publicly verifiable audit log of all successful crawl events, a key feature distinguishing the protocol from opaque, centralized alternatives.[1]

- **Purpose:** The ledger is called by the publisher's edge worker *after* a payment has been verified and content has been served. It provides the "proof of crawl" that completes the interaction record.
- **Structs:**
    - struct CrawlEvent { address crawlerWallet; address publisherWallet; bytes32 contentHash; uint256 timestamp; }: A data structure to hold the details of each logged event. The contentHash is a keccak256 hash of the crawled URL to provide verifiability while preserving a degree of privacy.
- **Events:**
    - event CrawlLogged(address indexed crawlerWallet, address indexed publisherWallet, bytes32 contentHash, uint256 timestamp): The core event that makes the audit trail discoverable and indexable by third-party services and the protocol's own dashboard. The crawlerWallet and publisherWallet are indexed for efficient querying.
- **State Variables:**
    - mapping(bytes32 => bool) private _loggedCrawls: A mapping to prevent replay attacks. The key is a keccak256 hash of the event parameters (crawlerWallet, publisherWallet, contentHash), ensuring that the exact same crawl event cannot be logged twice.
- **Core Functions:**
    - logCrawl(address crawlerWallet, bytes32 contentHash):
        - **Access Control:** This function will be permissioned. For the MVP, it will only be callable by a trusted "forwarder" address controlled by the protocol team to simplify the initial deployment. In a future version, publishers would be granted direct permission to call it.
        - **Logic:** The function identifies the publisherWallet from msg.sender. It then computes the unique event hash and checks the _loggedCrawls mapping. If the event has not been logged, it marks it as logged and emits the CrawlLogged event with the relevant data, including the current block timestamp.

The following table provides a high-level specification for each smart contract in the MVP, serving as a blueprint for development.

Table 2: MVP Smart Contract Inventory

Contract Name	Standard	Purpose	Key Public/External Functions	Key Events
CrawlNFT.sol	ERC-721, Ownable	On-chain anchor for publisher's ToS. One per site.	constructor(owner, termsURI), getTermsURI()	Transfer (from OpenZeppelin)
PaymentProcessor.sol	Custom	Receives USDC payments from crawlers and forwards them to publishers.	receivePayment(publisherWallet, amount)	PaymentReceived(publisher, crawler, amount)
ProofOfCrawlLedger.sol	Custom	Immutable, on-chain audit log of successful crawl events.	logCrawl(crawlerWallet, contentHash)	CrawlLogged(crawler, publisher, contentHash, timestamp)

Section 4: Core Component Deep Dive: The Publisher Gateway

The Publisher Gateway is the critical off-chain component that acts as the protocol's enforcement arm. Deployed as a serverless function on a publisher's content delivery network (CDN), it intercepts incoming requests, issues payment challenges, and verifies payment proofs. Its design is paramount to the system's function, as it is the active agent that translates the passive on-chain rules into real-time web interactions. This section details its implementation using Cloudflare Workers and TypeScript.

4.1 Implementation using Cloudflare Workers & TypeScript

The gateway will be built as a Cloudflare Worker to leverage its global distribution, low latency, and seamless integration with a large portion of the web.[1] The project will provide a starter template using TypeScript to ensure type safety and maintainability.[16]
The core logic resides within the fetch handler of the worker script, which is the entry point for every incoming HTTP request. The project will provide a template repository with a pre-configured wrangler.toml file and TypeScript environment, including generated types via wrangler types to provide strong typing for environment variables and bindings.[17]

TypeScript

```
// src/index.ts

// Interface for environment variables defined in wrangler.toml
export interface Env {
  PUBLISHER_WALLET_ADDRESS: string;
  PAYMENT_PROCESSOR_ADDRESS: string;
  CRAWL_PRICE_USDC: string;
  BASE_RPC_URL: string;
}

export default {
  async fetch(request: Request, env: Env, ctx: ExecutionContext): Promise<Response> {
    // 1. Identify if the request is from a known AI crawler.
    // 2. Check for a valid payment proof in the Authorization header.
    // 3. If crawler and no proof, issue a 402 challenge.
    // 4. If crawler with proof, verify the proof on-chain.
    // 5. If verification succeeds, serve content and log the crawl.
    // 6. If not a crawler, or verification fails, pass through or block.
    //... implementation details follow...
  }
};
```

4.2 Request Interception and Crawler Identification

The first step within the fetch handler is to determine if the request originates from an AI crawler that should be subject to the protocol. For the MVP, identification will be based on a combination of simple, pragmatic methods:

- **User-Agent Matching:** The worker will check the User-Agent header against a list of known crawler strings (e.g., GPTBot, ClaudeBot, PerplexityBot).[1] While this is easily spoofable, it serves as a necessary first-pass filter.
- **IP Address Range:** A maintained list of known IP address ranges for major AI companies will be used as a secondary check.
- **Future Enhancement:** In the full protocol, the primary identifier will be a Decentralized Identifier (DID) passed in a custom request header, which can be cryptographically verified.

4.3 Generating the HTTP 402 Response

If the worker identifies a request as coming from a known crawler and finds no valid payment proof, it must generate and return an HTTP 402 Payment Required response. This response is not just an error; it is the protocol's machine-readable invoice.
The implementation will use the standard Response constructor available in the Cloudflare Workers API to set the custom status code and the necessary x402 headers.[39] Cloudflare's platform fully supports the use of non-standard 4xx status codes like 402.[31]

TypeScript

```
// Inside the fetch handler, if a crawler is detected without payment proof:

const paymentDetails = {
  price: env.CRAWL_PRICE_USDC,
  recipient: env.PAYMENT_PROCESSOR_ADDRESS,
  network: 'base',
  token: 'USDC' // Or the specific token contract address on Base
};

const headers = new Headers({
  'Content-Type': 'application/json',
  // Following the x402 standard [1, 32]
  'x402-price': paymentDetails.price,
  'x402-recipient': paymentDetails.recipient,
  'x402-network': paymentDetails.network,
  'x402-token': paymentDetails.token
});

return new Response(JSON.stringify({ error: 'Payment Required' }), {
  status: 402,
  statusText: 'Payment Required',
  headers: headers
});
```

4.4 Verifying the Payment Proof

When the crawler makes its second request, it will include the proof of payment in the

Authorization header (e.g., Authorization: Bearer 0x...). The gateway must verify this proof by querying the Base blockchain.

Verification Steps:

1. **Extract Proof:** The worker parses the Authorization header to get the transaction hash.
2. **Initialize RPC Client:** It initializes a Viem Public Client, configured with the BASE_RPC_URL stored as a worker secret. This allows it to communicate with the Base network. For the MVP, this can be a free endpoint from Coinbase Developer Platform.[34]
3. **Fetch Transaction Receipt:** The worker calls publicClient.getTransactionReceipt({ hash: tx_hash }).[35] This is an asynchronous network call.
4. **Validate Receipt:** Upon receiving the receipt, the worker performs a series of critical validations:
    - It checks that a receipt was found and that receipt.status === 'success' to ensure the transaction did not revert.[35]
    - It parses the receipt.logs to find the Transfer event from the USDC contract.
    - It verifies that the to address in the transfer event matches the PAYMENT_PROCESSOR_ADDRESS.
    - It verifies that the value of the transfer is greater than or equal to the required CRAWL_PRICE_USDC.
5. **Serve Content:** If all checks pass, the worker fetches the original request against the origin server, allowing the actual content to be retrieved, and streams the response back to the crawler.
6. **Log Asynchronously:** To avoid delaying the content delivery, the worker wraps the final logCrawl transaction in ctx.waitUntil(). This Cloudflare Workers API method allows an asynchronous task to continue executing after the response has been sent to the client. The worker will then call the ProofOfCrawlLedger contract to immutably log the successful interaction.

4.5 Publisher Onboarding and Configuration

To meet the low-friction requirement, the configuration for the MVP will be managed entirely through the wrangler.toml file and Cloudflare's secrets interface. The provided template will instruct publishers to set the following environment variables and secrets:

- PUBLISHER_WALLET_ADDRESS
- CRAWL_NFT_ADDRESS
- PAYMENT_PROCESSOR_ADDRESS
- PROOF_OF_CRAWL_LEDGER_ADDRESS
- CRAWL_PRICE_USDC (e.g., a string like "0.001")
- BASE_RPC_URL (a secret containing the RPC endpoint URL)

Section 5: Core Component Deep Dive: The AI Crawler

Client & SDK

The AI Crawler Client SDK is the product offered to the demand side of the marketplace. Its quality, ease of use, and robustness will be a primary driver of adoption by AI companies. The design philosophy must be centered on abstracting away the formidable complexity of the underlying Web3 interactions.

5.1 SDK Design Philosophy

- **Abstraction is the Product:** The target user, an ML engineer, is an expert in data processing, not blockchain protocols. The SDK's primary value proposition is to make on-chain payments disappear into a familiar programming model. The developer should interact with a high-level, asynchronous API, such as crawler.fetch(url), and the SDK should handle the entire stateful 402 payment negotiation under the hood.
- **Internal State Management:** The SDK will be a sophisticated state machine, managing the lifecycle of a request internally. It will transition between states such as IDLE, AWAITING_PAYMENT_CHALLENGE, BUILDING_USER_OP, AWAITING_CONFIRMATION, and PAID. This internal management is invisible to the end-user, who simply awaits the final content response.
- **Extensibility and "Eject Button":** While the default API will be highly abstract, the SDK should provide escape hatches for advanced users. This includes exposing the underlying UserOperation builder and the Viem client, allowing power users to customize gas settings, inspect operations before signing, or integrate with their own infrastructure.

5.2 ERC-4337 UserOperation Builder

This is the technical core of the SDK. When a fetch call results in a 402 response, the SDK must programmatically construct, sign, and dispatch a valid UserOperation. This process will be managed using a dedicated library like Alchemy's aa-sdk or permissionless.js to handle the low-level details.[23]

The UserOperation Construction Flow:

1. **Parse 402 Headers:** The SDK extracts price, recipient, network, and token from the x402 response headers.
2. **Instantiate Builder:** It uses a high-level builder from the chosen AA library (e.g., Alchemy's AlchemyProvider [24]).
3. **Construct callData:** This is the most critical step. The goal is to make the smart wallet transfer USDC. The SDK will encode a function call to the smart wallet's generic execute function. The arguments to execute will be:

    - target: The address of the USDC contract on Base.
    - value: 0 (since no native ETH is being sent).
    - data: The encoded function call for USDC.transfer(recipient, amount), where recipient is the PaymentProcessor address and amount is the price from the 402 response.
4. **Estimate Gas:** The SDK will then make a call to the Bundler's eth_estimateUserOperationGas RPC method, passing the partially constructed UserOperation. The Bundler simulates the transaction and returns the required gas limits (callGasLimit, verificationGasLimit, preVerificationGas).[43]
5. **Populate paymasterAndData:** If the AI company has configured a gas sponsorship policy, the SDK will make an API call to the Paymaster service (e.g., Alchemy's Gas Manager). The Paymaster validates the operation against its policies and returns the necessary paymasterAndData string, which acts as its promise to pay the gas fee.[20] If no sponsorship is used, this field is left empty.
6. **Sign and Dispatch:** With all fields populated, the SDK hashes the UserOperation according to the EIP-4337 specification. This hash is then signed by the crawler's underlying private key. The resulting signature is added to the UserOperation, which is then sent to the Bundler's eth_sendUserOperation RPC method.

5.3 AI Crawler Client: UserOperation Construction Logic Table

This table provides a deterministic mapping from the protocol's inputs to the technical fields of an ERC-4337 UserOperation, serving as a clear specification for the SDK's implementation.

UserOperation Field	Source of Data	SDK Logic
sender	Crawler's configuration	The pre-known address of the crawler's smart wallet.
nonce	EntryPoint contract	Call entryPoint.getNonce(sender, 0) via the Bundler RPC.
initCode	N/A (for MVP)	0x (empty bytes), as the smart wallet account is assumed to be pre-deployed.
callData	402 Headers & Crawler Config	Encode a call to smartWallet.execute(USDC_CONTRACT_ADDRESS, 0, encoded_usdc_transfer_call).
Gas Parameters	Bundler RPC	Automatically call bundler.eth_estimateUserOperationGas with the partial UserOperation.
paymasterAndData	Paymaster Service API	If gas sponsorship is enabled,

		call the Paymaster's API to get this value. Otherwise, 0x.
signature	Crawler's private key	Hash the complete UserOperation (with other fields populated) and sign it with the wallet's owner key.

5.4 Wallet and Key Management Strategy

A critical operational challenge for AI companies will be securely managing the private key that controls the crawler's smart wallet. A naive approach of storing the raw key on every crawler instance would be a major security risk.
Recommended Strategy: The SDK will be designed to support a more robust, enterprise-grade key management architecture. AI companies will be advised to store the master private key for the smart wallet's owner in a secure, centralized keystore such as AWS KMS, Google Cloud KMS, or HashiCorp Vault. The individual crawler instances will not have direct access to this key. Instead, they will be granted IAM roles that allow them to make authenticated API calls to a secure internal signing service. When the SDK needs to sign a UserOperation, it will send the hash to this service, which will use the master key to generate the signature and return it. This architecture centralizes the key, minimizes its exposure, and allows for robust auditing and access control. The ERC-4337 smart wallet itself provides further layers of security, such as programmable spending limits, whitelists, and social recovery mechanisms, which are vastly superior to a standard Externally Owned Account (EOA).[18]

Section 6: Security Posture: A Continuous Audit & Defense-in-Depth Strategy

Given the financial nature of the protocol and the legal novelty of its on-chain licensing model, a comprehensive and multi-layered security strategy is not an optional extra but a foundational requirement for building trust and achieving adoption. The approach will be to "shift left," integrating security practices into every stage of the development lifecycle, from the first line of code to post-deployment monitoring.[45] This strategy will heavily leverage modern automated and AI-powered tools to create a robust, continuous audit process.

6.1 The "Shift Left" Security Philosophy

The core principle is to detect and remediate vulnerabilities as early as possible in the development process. A bug found in a developer's local environment is orders of magnitude cheaper and faster to fix than one discovered in a production system after an exploit.[45] This requires a culture of security and the integration of automated security tools directly into the developer workflow and CI/CD pipeline.

6.2 Development Lifecycle Security (CI/CD Integration)

The Continuous Integration/Continuous Deployment (CI/CD) pipeline will be the first line of automated defense.

- **Static Analysis (SAST):**
    - **Linting:** On every git commit, a pre-commit hook will run a linter like **Solhint**.[46] This provides immediate feedback to developers on stylistic issues, code complexity, and basic security anti-patterns.
    - **Deep Static Analysis:** On every pull request submitted to the main development branch, the CI pipeline will automatically execute a scan with **Slither**.[47] Slither is an industry-standard static analysis framework that performs a deep analysis of the smart contract's control flow and data flow to detect a wide range of common vulnerabilities, such as reentrancy, integer overflows/underflows, and access control issues, with a low false-positive rate.[47] The pull request will be blocked from merging if Slither reports any medium or high-severity vulnerabilities.
- **AI-Powered Scanning:**
    - In parallel with Slither, the CI pipeline will also trigger a scan using an AI-powered security tool like **SolidityScan** or **AuditBase**.[50] These tools leverage large language models trained on thousands of audited smart contracts and known exploits. They excel at identifying more complex or nuanced vulnerabilities, including potential business logic flaws, that rule-based static analyzers might miss.[51] Integrating a basic scan from one of these services into the CI pipeline provides an additional, powerful layer of automated defense.

6.3 Pre-Deployment Security

Before any new version of the smart contracts is deployed to a public network, it must pass a more intensive gauntlet of security checks.

- **Deep AI-Assisted Audit:** A comprehensive "Deep Scan" will be commissioned from a service like **Nethermind's AuditAgent**.[53] Unlike the quick CI scan, this process uses significantly more computational resources to perform advanced analysis, including simulating a range of potential attack scenarios and modeling the behavior of malicious actors.

- **Automated Fuzzing: Foundry's** powerful, built-in fuzz testing capabilities will be used extensively.[10] Critical functions, especially those in PaymentProcessor.sol that handle funds, will be subjected to thousands of iterations with randomized inputs. This is highly effective at uncovering unexpected edge cases, such as integer overflows or state corruption, that might not be covered by standard unit tests.
- **Formal Third-Party Manual Audit:** Automated tools are necessary but not sufficient. For the main V1.0 launch and any subsequent major upgrades, a full manual audit by a reputable, independent security firm (e.g., Consensys Diligence, OpenZeppelin, Trail of Bits) is non-negotiable.[32] Human auditors are essential for understanding the protocol's intent and identifying sophisticated business logic flaws, economic exploits, or novel attack vectors that automated tools are not designed to find.[52]

6.4 Post-Deployment & Continuous Monitoring

Security is an ongoing process, not a one-time event. The protocol must be architected for resilience and monitored continuously after deployment.

- **Architecting for Incident Response:**
  - **Emergency Stop (Pausability):** All V1.0 (post-MVP) contracts handling funds or critical state changes will inherit from OpenZeppelin's Pausable.sol utility.[37] This implements an "emergency stop" mechanism, allowing a designated admin (initially a multi-signature wallet held by the core team) to halt all critical functions in the event of a detected exploit, buying time to assess and respond.
  - **Upgradability:** The V1.0 contracts will be deployed using the UUPS (Universal Upgradeable Proxy Standard) proxy pattern. This allows the logic contract to be upgraded to patch bugs or add features without requiring a complex and risky data migration from the old contracts to new ones.[37]
- **On-Chain Monitoring:**
  - An on-chain monitoring service, such as **Forta**, will be deployed. Forta uses a network of independent detection bots to monitor transactions and state changes in real-time and emit alerts on suspicious activity.[54]
  - Custom detection bots will be developed to watch for protocol-specific anomalies, including: a sudden, abnormally high volume of payments to a single publisher; a rapid succession of logCrawl events from a single crawler; or unexpected reverts from the PaymentProcessor contract, which could indicate an exploit attempt.[54]
- **Incident Response Plan (IRP):** A formal IRP will be documented and rehearsed.[37] This plan will outline the precise steps to be taken upon receiving a critical alert from the monitoring system:
  1. **Triage:** A designated on-call engineer assesses the alert's severity.
  2. **Escalate:** If critical, the core team is convened via a secure communication

channel.

3. **Halt:** The emergency stop is triggered via the admin multi-sig to prevent further damage.
4. **Communicate:** A public statement is prepared to inform the community of the situation.
5. **Remediate:** A patch is developed, tested, audited, and deployed via the upgrade mechanism.

The following table codifies this defense-in-depth strategy into an actionable workflow for the engineering team.

Table 3: Multi-Layered Security & Audit Workflow

Development Phase	Action / Tool	Purpose	Key Research Snippets
Local Development	Pre-commit Hook w/ Solhint	Catch basic style and security issues before code is ever committed.	[46]
Continuous Integration (CI)	Slither Static Analysis	Automated detection of common vulnerabilities on every PR.	[47]
Continuous Integration (CI)	AI Scan (e.g., SolidityScan Basic)	Automated detection of more complex issues using AI models.	[50]
Pre-Deployment	Foundry Fuzz Testing	Uncover edge-case bugs by bombarding functions with random data.	[10]
Pre-Deployment	Deep AI Audit (e.g., AuditAgent Deep)	In-depth, AI-driven analysis and attack simulation.	[51]
Pre-Deployment	Manual Third-Party Audit	Expert human review to find business logic and novel vulnerabilities.	[32]
Post-Deployment	On-Chain Monitoring (e.g., Forta)	Real-time detection of anomalous transactions and potential exploits.	[54]
Post-Deployment	Incident Response Plan (IRP)	A clear plan for pausing contracts and managing a live incident.	[37]

Section 7: Roadmap to a Robust System: Scaling, Decentralization, and Advanced Features

The MVP is designed to be a focused starting point, not the final destination. The long-term vision for the Pay-Per-Crawl protocol is to evolve from a centralized product into a decentralized, community-owned piece of public infrastructure for the AI-driven web. This roadmap outlines a deliberate, phased journey to achieve that vision, tackling product, market, and governance risks in a logical sequence.

7.1 Phase 1: MVP & Atomic Network (Months 1-6)

- **Primary Goal:** To rigorously validate the core economic loop and the underlying legal framework in a real-world environment.
- **Technical Milestones:**
    - Successfully develop and deploy all MVP components as specified in Sections 3, 4, and 5. This includes the core smart contracts, the publisher gateway, the AI crawler SDK, and the publisher onboarding web application.
    - Pass a focused, MVP-scoped security review, including comprehensive static analysis (Slither), AI-powered scanning, and fuzz testing.
    - Execute the go-to-market strategy outlined in the viability analysis by identifying and onboarding an "atomic network" of 10-20 high-value publishers within a single, well-defined content vertical.[1]
    - Successfully process the first 100,000 paid crawl transactions, generating tangible on-chain data to prove the model's viability to future partners and investors.

7.2 Phase 2: V1.0 - Liquid Data Markets (Months 7-12)

- **Primary Goal:** To evolve beyond simple per-request payments by introducing liquidity and sophisticated pricing mechanisms, transforming data access rights into a tradable, liquid asset class. This is a key differentiator that centralized competitors cannot easily replicate.[1]
- **Technical Milestones:**
    - **CrawlToken.sol (ERC-20):** Develop and deploy a standard ERC-20 contract to represent fungible, pre-paid rights to access content. Each publisher's Crawl-NFT will govern its own unique CrawlToken.
    - **CrawlTokenMinter.sol:** Create a new smart contract, controlled by the

publisher's Crawl-NFT, that allows them to mint new batches of their CrawlTokens (e.g., minting 1,000,000 tokens, each redeemable for one crawl).
    - **Protocol Upgrade:** Upgrade the core protocol contracts (using the UUPS proxy pattern) to accept CrawlTokens as a payment method alongside USDC. A crawler can now choose to pay per-request with stablecoins or redeem a pre-purchased CrawlToken.
    - **DEX Integration:** Actively encourage and facilitate the creation of liquidity pools on Base-native decentralized exchanges (DEXs) like Uniswap for various CrawlToken/USDC pairs. This enables dynamic, market-driven price discovery for data access rights, allowing for speculation and hedging.[1]
    - **Publisher Dashboard V2:** Enhance the publisher dashboard with tools for minting and managing CrawlTokens, as well as analytics for viewing their token's on-market trading activity.
    - **Full Third-Party Audit:** Commission a comprehensive manual security audit of the entire V1.0 smart contract suite, including the new token contracts and the upgraded payment processor.

7.3 Phase 3: V2.0 - Full Decentralization & Governance (Months 13+)

- **Primary Goal:** To complete the transition from a company-controlled product to a credibly neutral, community-owned public utility, ensuring its long-term resilience and alignment with the open web ethos.
- **Technical Milestones:**
    - **Governance Token Launch:** Design, create, and launch a native governance token for the protocol. The token distribution will be strategically designed to empower all key stakeholders, including retroactive airdrops to early-adopting publishers and crawlers, grants for ecosystem developers, and a vested allocation for the founding team.[1]
    - **Protocol DAO Deployment:**
        - Establish a Decentralized Autonomous Organization (DAO) using a battle-tested on-chain governance framework, such as OpenZeppelin's implementation of Governor Bravo.
        - Deploy a community-governed Treasury contract that will receive a small, protocol-wide fee from all transactions.
        - Implement a robust governance process, likely a hybrid model using gas-less off-chain voting on platforms like Snapshot for signaling and debate, followed by binding on-chain votes for executing major decisions like protocol upgrades or treasury spending.[1]
    - **Progressive Decentralization:** Execute a formal handover of control. The ownership and administrative privileges of the core protocol contracts (e.g., the ability to set protocol fees or upgrade implementations) will be transferred from

the founding team's multi-signature wallet to the DAO's governance contract.
    - **Permissionless Ecosystem:** Fully automate and open the onboarding process, allowing any publisher or AI company to integrate with the protocol without requiring permission from the core team.

7.4 Future Features & Research Directions

Beyond V2.0, several avenues for further innovation exist:

- **Dynamic and Algorithmic Pricing:** Enable publishers to deploy smart contracts that implement dynamic pricing models, where the cost per crawl could adjust based on demand, content age, or other on-chain signals.
- **Reputation Systems:** Develop on-chain reputation systems, potentially using non-transferable NFTs ("Soulbound Tokens"), to score the quality of publishers and the behavior of crawlers, enabling a market for trusted data sources.
- **Cross-Chain Expansion:** Deploy the protocol on other high-throughput, low-cost L2 networks such as Arbitrum, Optimism, or Polygon to expand the addressable market of publishers and AI companies.[4]
- **Privacy-Preserving Crawls:** Research and implement solutions using Zero-Knowledge Proofs (ZKPs) that would allow a crawler to prove to a publisher that it has paid for access without revealing its specific on-chain identity for every single crawl, enhancing privacy for AI companies.

The following table provides a high-level strategic overview of the project's phased evolution.

Table 4: Phased Development Roadmap (MVP to V2.0)

Phase	Name	Primary Goal	Key Technical Features	Target Outcome
1 (Months 1-6)	MVP: The Atomic Transaction	Validate the core economic loop & legal framework.	Crawl-NFT, Per-request USDC payments, Proof-of-Crawl ledger, Edge Worker, Crawler SDK.	Product-market fit with a niche "atomic network."
2 (Months 7-12)	V1.0: Liquid Data Markets	Introduce liquidity and dynamic price discovery.	ERC-20 CrawlTokens, Token Minter, DEX integration, Upgradable contracts.	A functioning, open market for data access rights.
3 (Months 13+)	V2.0: Protocol-as-Public-Good	Achieve credible neutrality and community	Governance Token, Protocol DAO, On-chain	A self-sustaining, decentralized protocol governed

		ownership.	Treasury, Permissionless onboarding.	by its stakeholders.

Works cited

1. Pay-Per-Crawl Protocol Viability Analysis
2. Vercel Functions, accessed July 3, 2025, https://vercel.com/docs/functions
3. What Is Base Crypto? A Guide to Coinbase's Layer 2 Solution - Ulam Labs, accessed July 3, 2025, https://www.ulam.io/blog/exploring-base-a-layer-2-blockchain-from-coinbase
4. Top Smart Contract Platforms: Choosing the Best Blockchain for Your Project in 2025 - ilink, accessed July 3, 2025, https://ilink.dev/blog/top-smart-contract-platforms-choosing-the-best-blockchain-for-your-project-in-2025/
5. Top 6 Smart Contract Languages to Learn in 2025 - Rapid Innovation, accessed July 3, 2025, https://www.rapidinnovation.io/post/top-6-smart-contract-programming-languages
6. Best Web3 Development Tools and Frameworks of 2025 - Debut Infotech, accessed July 3, 2025, https://www.debutinfotech.com/blog/best-web3-development-tools
7. Error Codes - Vercel, accessed July 3, 2025, https://vercel.com/docs/errors
8. DEPLOYMENT_DISABLED - Vercel, accessed July 3, 2025, https://vercel.com/docs/errors/DEPLOYMENT_DISABLED
9. Smart Contract Development Trends for 2025 - Blockchain Solutions, accessed July 3, 2025, https://blockchainsolutions.com.sa/blog/smart-contract-development-trends-2025/
10. Foundry Vs Hardhat - EatTheBlocks, accessed July 3, 2025, https://eattheblocks.com/foundry-vs-hardhat/
11. Introduction - Ethereum Blockchain Developer, accessed July 3, 2025, https://www.ethereum-blockchain-developer.com/advanced-mini-courses/remix-vs-truffle-vs-hardhat-vs-foundry
12. Foundry or hardhat : r/solidity - Reddit, accessed July 3, 2025, https://www.reddit.com/r/solidity/comments/1hwirns/foundry_or_hardhat/
13. Hardhat vs. Foundry: Which Is Better, or Should You Use Them Hand in Hand?, accessed July 3, 2025, https://dev.to/ceasermikes002/hardhat-vs-foundry-which-is-better-or-should-you-use-them-hand-in-hand-1400
14. Viem - Wagmi, accessed July 3, 2025, https://wagmi.sh/core/guides/viem
15. Wagmi | Reactivity for Ethereum apps, accessed July 3, 2025, https://wagmi.sh/
16. TypeScript · Cloudflare Pages docs, accessed July 3, 2025, https://developers.cloudflare.com/pages/functions/typescript/
17. Write Cloudflare Workers in TypeScript, accessed July 3, 2025,

https://developers.cloudflare.com/workers/languages/typescript/
18. Account Abstraction and ERC-4337 - Part 1 | QuickNode Guides, accessed July 3, 2025, https://www.quicknode.com/guides/ethereum-development/wallets/account-abstraction-and-erc-4337
19. Gelato's Guide to Account Abstraction: from ERC4337 to EIP7702, accessed July 3, 2025, https://gelato.network/blog/gelato-s-guide-to-account-abstraction-from-erc-4337-to-eip-7702
20. What are Paymasters? (ERC-4337) - Alchemy, accessed July 3, 2025, https://www.alchemy.com/overviews/what-is-a-paymaster
21. A deep dive into the main components of ERC-4337: Account Abstraction Using Alt Mempool — Part 4 | by Antonio Viggiano | Oak Security | Medium, accessed July 3, 2025, https://medium.com/oak-security/a-deep-dive-into-the-main-components-of-erc-4337-account-abstraction-using-alt-mempool-part-4-ab7dacbf64d4
22. Introducing Account Kit - Alchemy, accessed July 3, 2025, https://www.alchemy.com/blog/introducing-account-kit
23. alchemyplatform/aa-sdk - Account Kit - GitHub, accessed July 3, 2025, https://github.com/alchemyplatform/aa-sdk
24. Particle Auth with Alchemy's Account Kit, accessed July 3, 2025, https://developers.particle.network/guides/integrations/partners/alchemy
25. Account Abstraction Wallets - Turnkey Docs, accessed July 3, 2025, https://docs.turnkey.com/reference/aa-wallets
26. medium.com, accessed July 3, 2025, https://medium.com/buildbear/hardhat-vs-foundry-learn-how-to-use-both-in-your-project-0b2714e6c337#:~:text=Hardhat%20is%20a%20Javascript%20framework,same%20language%20as%20the%20code.
27. UserOperation – ERC 4337 Documentation, accessed July 3, 2025, https://www.erc4337.io/docs/understanding-ERC-4337/user-operation
28. Developer SDKs - Stackup, accessed July 3, 2025, https://docs.stackup.sh/docs/erc-4337-developer-sdks
29. How to connect multiple wallets to your React dApp using Web3Modal - DEV Community, accessed July 3, 2025, https://dev.to/sadiqful/how-to-connect-multiple-wallets-to-your-react-dapp-using-web3modal-5cmn
30. rainbow-me/rainbowkit-examples - GitHub, accessed July 3, 2025, https://github.com/rainbow-me/rainbowkit-examples
31. 402 Payment Required - HTTP - MDN Web Docs - Mozilla, accessed July 3, 2025, https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/402
32. A Guide to Smart Contract Security | Hedera, accessed July 3, 2025, https://hedera.com/learning/smart-contracts/smart-contract-security
33. Use EIP-4337 with Stackup SDK: create UserOperation and execute it sponsored by a PayMaster | by Miquel Cabot | Medium, accessed July 3, 2025, https://medium.com/@miquelcabot/use-eip-4337-with-stackup-sdk-create-user

operation-and-execute-it-sponsored-by-a-paymaster-494a2f9f828f
34. Verify a Smart Contract using Basescan API - Base Documentation, accessed July 3, 2025, https://docs.base.org/learn/foundry/verify-contract-with-basescan
35. getTransactionReceipt - Viem, accessed July 3, 2025, https://viem.sh/docs/actions/public/getTransactionReceipt.html
36. getTransactionReceipt - viem, accessed July 3, 2025, https://v1.viem.sh/docs/actions/public/getTransactionReceipt.html
37. Smart Contract Security in 2023: A Simple Checklist - Read More, accessed July 3, 2025, https://www.contractlogix.com/contract-management/smart-contract-security/
38. Write to Contract | Wagmi, accessed July 3, 2025, https://wagmi.sh/react/guides/write-to-contract
39. Response · Cloudflare Workers docs, accessed July 3, 2025, https://developers.cloudflare.com/workers/runtime-apis/response/
40. 4xx Client Error · Cloudflare Support docs, accessed July 3, 2025, https://developers.cloudflare.com/support/troubleshooting/http-status-codes/4xx-client-error/
41. How can I know a hash mined and confirmed by ethers.js - Ethereum Stack Exchange, accessed July 3, 2025, https://ethereum.stackexchange.com/questions/80617/how-can-i-know-a-hash-mined-and-confirmed-by-ethers-js
42. Pimlico Docs: Home, accessed July 3, 2025, https://docs.pimlico.io/
43. How ERC-4337 Gas Estimation Works - Alchemy, accessed July 3, 2025, https://www.alchemy.com/blog/erc-4337-gas-estimation
44. State of Wallets - Part 2: Smart Accounts (Account Abstraction – From Theory to Practice), accessed July 3, 2025, https://www.coinbase.com/blog/state-of-wallets-2
45. Consensys Diligence: Smart Contract Audits, accessed July 3, 2025, https://diligence.consensys.io/
46. Slither - Solidity Tools - Alchemy, accessed July 3, 2025, https://www.alchemy.com/dapps/slither
47. Slither | Discover and Try Web3 Tools, accessed July 3, 2025, https://tryweb3.tools/tool/?name=Slither
48. crytic/slither: Static Analyzer for Solidity and Vyper - GitHub, accessed July 3, 2025, https://github.com/crytic/slither
49. Slither: A Leading Static Analyzer for Smart Contracts - 101 Blockchains, accessed July 3, 2025, https://101blockchains.com/slither-tutorial/
50. SolidityScan: Best Smart Contract Scanner & Auditing Tool, accessed July 3, 2025, https://solidityscan.com/
51. Leading AI Smart Contract Audit Services for Enhanced Security - AuditBase, accessed July 3, 2025, https://www.auditbase.com/solutions/ai-smart-contract-audit
52. AI-Assisted Security Audits. A Practical Guide with Real-World... | by Eduard Kotysh - Medium, accessed July 3, 2025, https://medium.com/oak-security/ai-assisted-security-audits-0bd76608e3be

53. AuditAgent, accessed July 3, 2025, https://auditagent.nethermind.io/
54. high level of security and continuous monitoring for analyzing smart contract behaviors, accessed July 3, 2025, https://www.researchgate.net/publication/391339756_HIGH_LEVEL_OF_SECURITY_AND_CONTINUOUS_MONITORING_FOR_ANALYZING_SMART_CONTRACT_BEHAVIORS
55. Solidity Security Practices Part V: Continuous Monitoring | by Kaan Kaçar | Coinmonks, accessed July 3, 2025, https://medium.com/coinmonks/solidity-security-practices-part-iv-continuous-monitoring-85129c3ba9ef
56. AI Smart-Contract Auditor | ChainGPT Documentation, accessed July 3, 2025, https://docs.chaingpt.org/ai-tools-and-applications/ai-smart-contract-auditor
57. eth_getTransactionReceipt | Ethereum - Chainstack Docs, accessed July 3, 2025, https://docs.chainstack.com/reference/ethereum-gettransactionreceipt