A Developer's Guide to Building the Tachi Pay-Per-Crawl Protocol MVP

Introduction: Architecting the Tachi Protocol MVP

This document provides a definitive, step-by-step technical execution plan for building the Minimum Viable Product (MVP) of the Tachi Pay-Per-Crawl protocol. The guide is intended for a technically proficient developer and will focus exclusively on implementation, leveraging GitHub Copilot as a primary coding agent. The objective is to produce a developer's playbook for creating a functional, end-to-end demonstration of the protocol's core value proposition. The MVP is composed of four primary, interacting systems, as defined in the foundational technical plans.[1] These systems work in concert to create a new economic layer for the web:

1. **The On-Chain Foundation:** A suite of Solidity smart contracts deployed on the Base Layer 2 network that govern licensing, payment, and auditing. This forms the immutable, trusted backbone of the entire protocol.
2. **The Publisher Application:** A Next.js web application that serves as the publisher's control panel for onboarding, configuration, and monitoring. This is the primary interface for the supply side of the marketplace.
3. **The Publisher Gateway:** A serverless Cloudflare Worker that acts as the protocol's real-time enforcement mechanism. It intercepts requests from AI crawlers, issues payment challenges, and verifies on-chain payments before granting access to content.
4. **The AI Crawler SDK:** A proof-of-concept Node.js library that enables an AI crawler to interact with the protocol, programmatically handle payment challenges using Account Abstraction, and access content in a compliant manner.

Development will adhere to a set of strict principles designed to maximize learning and minimize time-to-market. The singular goal is to validate the core economic loop of the protocol. This involves prioritizing the core problem of executing a single, atomic pay-per-crawl transaction, ruthlessly minimizing integration friction for publishers, optimizing the developer experience for AI crawler engineers, and strategically deferring full decentralization features like on-chain governance in favor of speed and MVP validation.[1]

Part I: The On-Chain Foundation: Smart Contracts

This part details the creation of the protocol's immutable logic on the Base blockchain. The design prioritizes security, gas efficiency, and clarity, forming the trusted backbone of the entire system. The contracts are intentionally simple and non-upgradable for the MVP to minimize the security attack surface and accelerate the audit process, a strategic decision that prioritizes market and legal validation over premature feature completeness.[1]

Section 1: Hybrid Development Environment Setup

The establishment of a robust and efficient development environment is the first critical step. The protocol's success requires both unimpeachable security for its on-chain components and sophisticated scripting for deployment and off-chain integration. A hybrid environment that leverages the distinct strengths of both Foundry and Hardhat is the optimal approach to meet these dual requirements.[1]
This model represents a deliberate separation of concerns. Foundry, with its exceptional performance and Solidity-native testing, is the superior tool for validating the correctness and security of the core contract logic itself.[1] Its built-in fuzzing capabilities are invaluable for discovering edge-case vulnerabilities.[1] Conversely, Hardhat's powerful JavaScript/TypeScript environment excels at tasks that bridge the on-chain and off-chain worlds. It is the ideal choice for writing complex deployment scripts that interact with environment variables and external APIs, and for running integration tests that simulate the entire system, including the Cloudflare Worker and AI Crawler SDK.[1] This section provides the precise commands and configurations to establish this "best of both worlds" setup.

AI Coding Agent Prompt

Code snippet

```
As a senior blockchain developer, generate a series of shell commands and configuration files to set up a hybrid Hardhat and Foundry project.

1.  **Project Initialization:** Provide commands to create a directory, initialize git, and initialize a pnpm project.
2.  **Install Dependencies:** Provide commands to install `hardhat`, `@nomicfoundation/hardhat-toolbox`, and `@nomicfoundation/hardhat-foundry`.
3.  **Hardhat Setup:** Provide the command to initialize an empty Hardhat project.
4.  **Foundry Integration:** Provide the command to initialize Foundry within the Hardhat project.
5.  **Library Installation:** Provide commands to install `@openzeppelin/contracts` for Hardhat (via pnpm) and for Foundry (via `forge install`).
6.  **`hardhat.config.ts`:** Generate a complete `hardhat.config.ts` file. It should import
```

`dotenv`, `@nomicfoundation/hardhat-toolbox`, and `@nomicfoundation/hardhat-foundry`. Configure it for the `base-sepolia` testnet, sourcing the RPC URL and a private key from a `.env` file. Use Solidity version `0.8.20` as specified in the technical plan.[1]
7. **`foundry.toml`:** Generate a `foundry.toml` file. It should be configured to align with Hardhat's directory structure (e.g., `src = 'contracts'`, `test = 'test'`, `libs = ['node_modules', 'lib']`).

Generated Commands and Configuration

Project Initialization and Dependency Installation

Execute the following commands in your terminal to create the project directory and install the necessary development dependencies.

Bash

```bash
# 1. Create project directory and initialize
mkdir tachi-protocol-contracts
cd tachi-protocol-contracts
git init
pnpm init

# 2. Install Hardhat and its core plugins
pnpm add -D hardhat @nomicfoundation/hardhat-toolbox
@nomicfoundation/hardhat-foundry dotenv

# 3. Initialize an empty Hardhat project
pnpm exec hardhat init
# When prompted, select: 'Create an empty hardhat.config.js'

# 4. Initialize Foundry within the Hardhat project
pnpm exec hardhat init-foundry
# This creates foundry.toml and installs forge-std in the 'lib' directory [3]

# 5. Install OpenZeppelin Contracts for both Hardhat and Foundry
pnpm add -D @openzeppelin/contracts
pnpm exec forge install OpenZeppelin/openzeppelin-contracts
```

hardhat.config.ts

This file configures Hardhat to use the correct Solidity version, connect to the Base Sepolia testnet, and load the Foundry plugin. The private key and RPC URL are loaded securely from a .env file, which should never be committed to version control.[4]

TypeScript

```
import { HardhatUserConfig } from "hardhat/config";
import "@nomicfoundation/hardhat-toolbox";
import "@nomicfoundation/hardhat-foundry";
import "dotenv/config";

const BASE_SEPOLIA_RPC_URL = process.env.BASE_SEPOLIA_RPC_URL |

| "https://sepolia.base.org";
const PRIVATE_KEY = process.env.PRIVATE_KEY |

| "0xkey";

const config: HardhatUserConfig = {
  solidity: "0.8.20",
  networks: {
    hardhat: {
      // Local development network
    },
    base_sepolia: {
      url: BASE_SEPOLIA_RPC_URL,
      accounts:,
      chainId: 84532,
    },
  },
  paths: {
    sources: "./contracts",
    tests: "./test",
    cache: "./cache",
    artifacts: "./artifacts",
  },
};
```

```
export default config;
```

foundry.toml

The init-foundry command creates a foundry.toml file. It should be modified to ensure Foundry knows where to find libraries installed via both forge (in lib) and pnpm (in node_modules), ensuring seamless interoperability.[6]

Ini, TOML

```
# foundry.toml
[profile.default]
src = 'contracts'
out = 'out'
libs = ['node_modules', 'lib']
test = 'test'
cache_path = 'cache_forge'

# See more config options
https://github.com/foundry-rs/foundry/blob/master/crates/config/README.md#all-options
```

Section 2: The License Anchor: CrawlNFT.sol

This smart contract, CrawlNFT.sol, is the on-chain anchor for each publisher's license. It is implemented as a non-fungible token (NFT) following the ERC-721 standard. Its primary purpose is not to be a tradable asset but to serve as a permanent, verifiable credential that immutably links a publisher's website to their governing Terms of Service.[1]
The design of this contract is a direct technical implementation of a legal primitive. By storing the termsOfServiceURI (an IPFS hash) at the time of minting, it creates an unchangeable record of the "Offer" being made by the publisher.[1] The subsequent decision to make the NFT non-transferable by overriding the
_beforeTokenTransfer hook is a critical security and legal consideration. It ensures that the on-chain license cannot be separated from the operational control of the website, preventing a scenario where the legal rights are sold or transferred independently of the content asset itself.[1] This architecture transforms a legal concept into a composable piece of Web3 infrastructure, forming the basis for a legally coherent electronic contract when combined with a crawler's payment.[1]

AI Coding Agent Prompt

Code snippet

```
Generate the Solidity code for a smart contract named `CrawlNFT.sol` using pragma version
`^0.8.20`.

The contract must:
1.  Inherit from OpenZeppelin's `ERC721.sol`, `Ownable.sol`, and `Counters.sol`.
2.  Be initialized with a name "Tachi Crawl License" and symbol "TACHIL".
3.  Use OpenZeppelin's `Counters.Counter` for managing token IDs.
4.  Have a mapping from `uint256` (tokenId) to `string` to store a `termsOfServiceURI`.
5.  Have a public `mint` function that:
    - Is callable only by the `owner`.
    - Takes a `publisher` address and a `termsURI` string as arguments.
    - Mints a new token to the `publisher` using an auto-incrementing token ID.
    - Stores the `termsURI` associated with the new token ID.
6.  Have a public view function `getTermsURI(uint256 tokenId)` that returns the `termsURI` for
a given token.
7.  Override the `_beforeTokenTransfer` function to `revert` on any attempt to transfer a token
after its initial minting (i.e., when `from` is not `address(0)`), effectively making the NFTs
non-transferable.
```

Generated Smart Contract: contracts/CrawlNFT.sol

Solidity

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

/**
```

```
 * @title CrawlNFT
 * @dev This contract represents a publisher's license as a non-transferable ERC-721 token.
 * It serves as an immutable on-chain anchor for the publisher's Terms of Service.
 */
contract CrawlNFT is ERC721, Ownable {
    using Counters for Counters.Counter;
    Counters.Counter private _tokenIdCounter;

    // Mapping from token ID to the URI of the terms of service
    mapping(uint256 => string) private _termsOfServiceURIs;

    /**
     * @dev Sets the token name and symbol.
     */
    constructor() ERC721("Tachi Crawl License", "TACHIL") Ownable(msg.sender) {}

    /**
     * @dev Mints a new license NFT to a publisher.
     * @param publisher The address of the publisher receiving the license.
     * @param termsURI The IPFS URI of the terms of service document.
     */
    function mint(address publisher, string memory termsURI) public onlyOwner {
        _tokenIdCounter.increment();
        uint256 tokenId = _tokenIdCounter.current();
        _safeMint(publisher, tokenId);
        _termsOfServiceURIs[tokenId] = termsURI;
    }

    /**
     * @dev Returns the terms of service URI for a given token ID.
     * @param tokenId The ID of the token.
     * @return The terms of service URI string.
     */
    function getTermsURI(uint256 tokenId) public view returns (string memory) {
        require(_exists(tokenId), "CrawlNFT: URI query for nonexistent token");
        return _termsOfServiceURIs[tokenId];
    }

    /**
     * @dev Hook that is called before any token transfer.
     * This override prevents the transfer of tokens after they are minted,
     * making them non-transferable soulbound-style credentials.
     */
    function _beforeTokenTransfer(address from, address to, uint256 firstTokenId, uint256
batchSize) internal virtual override {
        super._beforeTokenTransfer(from, to, firstTokenId, batchSize);

        if (from!= address(0)) {
            revert("CrawlNFT: This license token is non-transferable.");
        }
    }
}
```

Section 3: The Payment Conduit: PaymentProcessor.sol

The PaymentProcessor.sol contract is the financial heart of the MVP. It is intentionally designed to be a simple, secure, and gas-efficient conduit for funds.[1] Its sole responsibility is to receive USDC payments from AI crawlers and immediately forward them to the designated publisher. This "pass-through" architecture is a deliberate security choice. By never holding funds itself, the contract minimizes its attack surface and avoids the complexities and risks associated with managing balances or handling native ETH, such as re-entrancy vulnerabilities.[7]
The contract relies on the trusted and audited SafeERC20 library from OpenZeppelin to handle all token interactions.[9] The
processPayment function uses a safeTransferFrom and safeTransfer pattern, ensuring that the entire financial operation—pulling funds from the crawler and pushing them to the publisher—occurs atomically within a single transaction. This design makes the contract maximally auditable and secure, functioning as a trusted on-chain router rather than a risky custodian of funds.[10]

AI Coding Agent Prompt

Code snippet

Generate the Solidity code for a smart contract named `PaymentProcessor.sol` using pragma version `^0.8.20`.

The contract must be a secure, stateless payment forwarder for USDC tokens. It should:
1. Import OpenZeppelin's `SafeERC20.sol` and `IERC20.sol`.
2. Use the `SafeERC20` library for the `IERC20` interface.

3. Have an immutable public state variable `usdcToken` of type `IERC20`.
4. The constructor should accept the address of the USDC token and set the `usdcToken` state variable.
5. Define a `PaymentReceived` event that logs the `crawlerWallet` (indexed), `publisherWallet` (indexed), and `amount`.
6. Have a single external function `processPayment(address publisherWallet, uint256 amount)`. This function must:
    - Securely transfer `amount` of USDC from the `msg.sender` (the crawler's wallet) to the `PaymentProcessor` contract itself.
    - Immediately transfer the same `amount` of USDC from the contract to the `publisherWallet`.
    - Emit the `PaymentReceived` event with the correct parameters (`msg.sender` as crawler, `publisherWallet`, and `amount`).

Generated Smart Contract: contracts/PaymentProcessor.sol

Solidity

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

/**
 * @title PaymentProcessor
 * @dev A stateless contract to securely process and forward USDC payments
 * from AI crawlers to publishers in a single, atomic transaction.
 */
contract PaymentProcessor {
    using SafeERC20 for IERC20;

    /**
     * @dev The address of the canonical USDC token contract.
     */
    IERC20 public immutable usdcToken;

    /**
     * @dev Emitted when a payment is successfully processed.
     */
    event PaymentReceived(
        address indexed crawlerWallet,
        address indexed publisherWallet,
        uint256 amount
    );

    /**
     * @dev Sets the USDC token address upon deployment.
     * @param usdcTokenAddress The address of the USDC contract on the target network.
     */
    constructor(address usdcTokenAddress) {
        usdcToken = IERC20(usdcTokenAddress);
    }

    /**
     * @dev Processes a payment from a crawler to a publisher.
     * The caller (crawler's smart wallet) must have approved this contract
     * to spend at least the `amount` of its USDC tokens.
     * @param publisherWallet The address of the publisher to receive the payment.
     * @param amount The amount of USDC to be transferred.
     */
    function processPayment(address publisherWallet, uint256 amount) external {
        require(publisherWallet!= address(0), "PaymentProcessor: Invalid publisher address");
        require(amount > 0, "PaymentProcessor: Payment amount must be greater than zero");

        // Pull the funds from the crawler's wallet to this contract
        usdcToken.safeTransferFrom(msg.sender, address(this), amount);

        // Immediately forward the funds to the publisher's wallet
        usdcToken.safeTransfer(publisherWallet, amount);

        emit PaymentReceived(msg.sender, publisherWallet, amount);
    }
}
```

Section 4: The Immutable Audit Trail: ProofOfCrawlLedger.sol

The ProofOfCrawlLedger.sol contract provides the immutable, transparent, and publicly verifiable audit log of all successful crawl events. This is a key feature that distinguishes the Tachi protocol from opaque, centralized alternatives where usage metering is a black box.[1]

This contract functions as the system's shared source of truth. By emitting a CrawlLogged event for every verified interaction, it creates a permanent, tamper-proof record that is accessible to all participants.[1] This has cascading benefits across the ecosystem. For publishers, it provides the verifiable data needed to populate their dashboards. For AI companies, it serves as a definitive audit trail of their data sourcing, which is invaluable for compliance with emerging regulations like the EU AI Act that mandate data provenance.[1] The design choice to index the
crawlerWallet and publisherWallet addresses in the event is critical, as it makes off-chain querying for this data highly efficient and scalable—a necessary feature for building responsive user-facing applications like the publisher dashboard. The inclusion of a replay protection mechanism ensures the integrity of the log, guaranteeing that each unique crawl can only be recorded once.[1]

AI Coding Agent Prompt

Code snippet

```
Generate the Solidity code for a smart contract named `ProofOfCrawlLedger.sol` using
pragma version `^0.8.20`.

The contract will serve as an on-chain audit log. It must:
1.  Import OpenZeppelin's `Ownable.sol`.
2.  Define a `CrawlLogged` event with indexed `crawlerWallet` (address) and
`publisherWallet` (address), and non-indexed `contentHash` (bytes32) and `timestamp`
(uint256).
3.  Include a mapping `_loggedCrawls` of type `mapping(bytes32 => bool)` to prevent replay
attacks.
4.  Have an external function `logCrawl(address publisherWallet, address crawlerWallet,
bytes32 contentHash)` that is `onlyOwner`. For the MVP, a single trusted entity calls this
function on behalf of publishers.
5.  The `logCrawl` function should:
    - Construct a unique identifier for the crawl event by computing
`keccak256(abi.encodePacked(publisherWallet, crawlerWallet, contentHash))`.
    - Check the `_loggedCrawls` mapping to ensure this event has not been logged before,
reverting with a "Crawl already logged" error if it has.
    - Mark the event as logged in the mapping by setting the corresponding value to `true`.
    - Emit the `CrawlLogged` event, using the function arguments for wallet addresses and
`contentHash`, and `block.timestamp` for the timestamp.
```

Generated Smart Contract: contracts/ProofOfCrawlLedger.sol

Solidity

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/access/Ownable.sol";

/**
 * @title ProofOfCrawlLedger
 * @dev An append-only ledger for recording verified crawl events, creating a
 * transparent and immutable audit trail for the protocol.
 */
contract ProofOfCrawlLedger is Ownable {
    /**
     * @dev Emitted when a successful crawl is logged.
     */
    event CrawlLogged(
        address indexed crawlerWallet,
        address indexed publisherWallet,
        bytes32 contentHash,
        uint256 timestamp
    );

    // Mapping to prevent replay attacks by ensuring each crawl is logged only once.
    mapping(bytes32 => bool) private _loggedCrawls;

    constructor() Ownable(msg.sender) {}

    /**
     * @dev Logs a verified crawl event. For the MVP, this is restricted to the owner
     * (a trusted forwarder service) to call on behalf of the publisher's gateway.
     * @param publisherWallet The wallet of the publisher whose content was crawled.
     * @param crawlerWallet The wallet of the AI crawler that paid for access.
     * @param contentHash A keccak256 hash of the crawled URL.
     */
    function logCrawl(
        address publisherWallet,
        address crawlerWallet,
        bytes32 contentHash
    ) external onlyOwner {
        bytes32 eventId = keccak256(abi.encodePacked(publisherWallet, crawlerWallet,
contentHash));

        require(!_loggedCrawls[eventId], "ProofOfCrawlLedger: Crawl already logged");

        _loggedCrawls[eventId] = true;

        emit CrawlLogged(
            crawlerWallet,
            publisherWallet,
            contentHash,
            block.timestamp
        );
    }
}
```

Section 5: Testing and Deployment to Base Sepolia

With the core contracts written, the next step is to ensure their robustness through comprehensive testing and then deploy them to a public testnet for integration with the off-chain components. This process involves both isolated unit tests for each contract and broader integration tests to verify their interactions.
A critical, practical hurdle in this phase is acquiring the necessary testnet assets. A developer cannot deploy contracts or simulate payments without testnet ETH for gas fees and testnet USDC for the payments themselves. Providing clear, direct links to reliable faucets is essential for a smooth developer experience and removes a common point of friction.[11] Furthermore, the deployment script must be configured with the correct, canonical contract address for USDC on the target testnet. An incorrect address would cause all payment transactions to fail. Including the verified address directly within the guide prevents this frustrating and time-consuming error.[13]

AI Coding Agent Prompts

Foundry Unit Test (CrawlNFT.t.sol)

Code snippet

Generate a Foundry test file in Solidity (`test/CrawlNFT.t.sol`) for the `CrawlNFT.sol` contract. It should use `forge-std/Test.sol`. Write a `setUp` function that deploys the contract. Write a test function `test_RevertsOnTransfer()` that:
1. Mints a token to a test address.
2. Uses `vm.prank` to switch the caller to the token owner.
3. Uses `vm.expectRevert` to assert that attempting to transfer the minted NFT to another address results in a revert with the message "CrawlNFT: This license token is non-transferable.".

Generated Foundry Test

Solidity

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "../contracts/CrawlNFT.sol";

contract CrawlNFTTest is Test {
    CrawlNFT public crawlNft;
    address public owner = address(0x1);
    address public publisher = address(0x2);
    address public recipient = address(0x3);

    function setUp() public {
        vm.prank(owner);
        crawlNft = new CrawlNFT();
    }

    function test_RevertsOnTransfer() public {
        // Mint token ID 1 to the publisher
        vm.prank(owner);
        crawlNft.mint(publisher, "ipfs://terms");
        // Expect a revert with the specific error message
        vm.expectRevert("CrawlNFT: This license token is non-transferable.");

        // Attempt to transfer the token as the publisher
        vm.prank(publisher);
        crawlNft.transferFrom(publisher, recipient, 1);
    }
}
```

Hardhat Deployment Script (scripts/deploy.ts)

Code snippet

Generate a Hardhat deployment script in TypeScript (`scripts/deploy.ts`). The script should:
1. Import `ethers` from Hardhat.
2. Define an `async main()` function.
3. Define the official Base Sepolia USDC contract address as a constant: `0x036CbD53842c5426634e7929541eC2318f3dCF7e`.
4. Use `ethers.getContractFactory` and the `deploy` method to deploy the `CrawlNFT`, `PaymentProcessor` (passing the USDC address to its constructor), and `ProofOfCrawlLedger` contracts.
5. Await the deployment of each contract.
6. Log the addresses of all deployed contracts to the console.
7. Include the standard Hardhat script execution and error handling boilerplate.

Generated Deployment Script

TypeScript

```
import { ethers } from "hardhat";

async function main() {
  const [deployer] = await ethers.getSigners();
  console.log("Deploying contracts with the account:", deployer.address);
  // The official USDC contract address on Base Sepolia Testnet [13, 14, 15]
  const usdcTokenAddress = "0x036CbD53842c5426634e7929541eC2318f3dCF7e";

  // Deploy CrawlNFT
  const CrawlNFT = await ethers.getContractFactory("CrawlNFT");
  const crawlNft = await CrawlNFT.deploy();
  await crawlNft.waitForDeployment();
  const crawlNftAddress = await crawlNft.getAddress();
  console.log(`CrawlNFT deployed to: ${crawlNftAddress}`);

  // Deploy PaymentProcessor
  const PaymentProcessor = await ethers.getContractFactory("PaymentProcessor");
  const paymentProcessor = await PaymentProcessor.deploy(usdcTokenAddress);
  await paymentProcessor.waitForDeployment();
  const paymentProcessorAddress = await paymentProcessor.getAddress();
  console.log(`PaymentProcessor deployed to: ${paymentProcessorAddress}`);

  // Deploy ProofOfCrawlLedger
  const ProofOfCrawlLedger = await ethers.getContractFactory("ProofOfCrawlLedger");
  const proofOfCrawlLedger = await ProofOfCrawlLedger.deploy();
  await proofOfCrawlLedger.waitForDeployment();
  const proofOfCrawlLedgerAddress = await proofOfCrawlLedger.getAddress();
  console.log(`ProofOfCrawlLedger deployed to: ${proofOfCrawlLedgerAddress}`);
}

main()
 .then(() => process.exit(0))
 .catch((error) => {
   console.error(error);
   process.exit(1);
  });
```

Base Sepolia Testnet Configuration Reference

To facilitate testing and deployment, the following table consolidates the essential configuration details and resources for the Base Sepolia testnet.

Parameter	Value / Link	Source(s)
Network Name	Base Sepolia	5
Chain ID	84532	N/A

Default RPC URL	https://sepolia.base.org	5
Block Explorer	https://sepolia.basescan.org	16
Official USDC Address	**0x036CbD53842c5426634e7929541eC2318f3dCF7e**	13
ETH Faucets	Coinbase Faucet Alchemy Faucet QuickNode Faucet	17
USDC Faucet	Circle Faucet (Select Base Sepolia)	12

Part II: The Publisher Application: A Next.js Front-End

This part details the construction of the publisher-facing MVP application. It is the primary interface for demonstrating the protocol's value, enabling publishers to seamlessly onboard, configure their license, and monitor tangible results on-chain. The application will be built using a modern stack comprising Next.js with the App Router, TypeScript, Tailwind CSS, and Shadcn/UI, ensuring a rapid development process and a polished, professional user experience.[1]

Section 6: Project Scaffolding and UI Setup

The foundation of the publisher application is a well-structured Next.js project. The choice of shadcn/ui is strategic; it is not a traditional component library but a tool that copies component source code directly into the project.[19] This provides the speed of using pre-built, accessible components while granting full ownership and control over the code, eliminating dependency bloat and making customization trivial. This approach aligns perfectly with the startup ethos of moving fast while maintaining complete control over the application's look, feel, and functionality.[1]

AI Coding Agent Prompt

Code snippet

Generate a series of shell commands to set up a new front-end project for the Tachi protocol.

1. Use `pnpm` and `create-next-app` to create a new Next.js project named `tachi-publisher-app`. Enable TypeScript, Tailwind CSS, ESLint, and the App Router. Use the

`src/` directory.
2. Navigate into the new directory.
3. Use the `shadcn-ui` CLI to initialize the UI library.
4. Use the `shadcn-ui` CLI to add the following components: `button`, `card`, `input`, `label`, `tabs`, `toast`, and `table`.

Generated Commands

Execute the following commands in your terminal to scaffold the Next.js project and install the necessary UI components.

Bash

```
# 1. Create the Next.js project
pnpm create next-app@latest tachi-publisher-app --typescript --tailwind --eslint --app
--src-dir

# 2. Navigate into the project directory
cd tachi-publisher-app

# 3. Initialize Shadcn/UI
# Follow the CLI prompts to configure paths and styles.
pnpm dlx shadcn-ui@latest init

# 4. Add the required components
pnpm dlx shadcn-ui@latest add button card input label tabs toast table
```

Section 7: Web3 Provider Configuration

To enable interaction with the blockchain, the application must be wrapped in the necessary Web3 context providers from wagmi and RainbowKit. The Web3 front-end ecosystem evolves rapidly, and using the latest, most stable patterns is crucial for avoiding dependency conflicts and ensuring a smooth developer experience.
The current canonical approach for wagmi@2.x and the latest version of RainbowKit involves using getDefaultConfig to configure the chains and WalletConnect projectId, and then wrapping the application in a specific hierarchy of providers: WagmiProvider, QueryClientProvider, and RainbowKitProvider.[20] This modern setup is simpler and more robust than older patterns and is essential for building a reliable dApp.

AI Coding Agent Prompt

Code snippet

```
Generate the code for a React client component in `src/app/providers.tsx` that sets up the
necessary Web3 providers for a Next.js App Router application.

The component, named `Web3Providers`, should:
1.  Import necessary components and functions from `@rainbow-me/rainbowkit`, `wagmi`,
`wagmi/chains`, and `@tanstack/react-query`.
2.  Import the RainbowKit CSS styles.
3.  Create a `QueryClient` instance.
4.  Use `getDefaultConfig` from RainbowKit to create a `config` object. This config should:
    - Set an `appName` to "Tachi Publisher App".
    - Use a `projectId` from an environment variable
`NEXT_PUBLIC_WALLETCONNECT_PROJECT_ID`.
    - Configure the `base` and `baseSepolia` chains.
    - Set `ssr: true`.
5.  The component should accept `children` as a prop.
6.  It should wrap the `children` in the following order: `WagmiProvider`,
`QueryClientProvider`, `RainbowKitProvider`.
```

Generated Provider Component: src/app/providers.tsx

This component centralizes all Web3 provider logic, keeping the root layout clean. It must be marked as a client component with "use client"; because it uses React hooks.

TypeScript

```typescript
"use client";

import "@rainbow-me/rainbowkit/styles.css";
import { getDefaultConfig, RainbowKitProvider } from "@rainbow-me/rainbowkit";
import { WagmiProvider } from "wagmi";
import { base, baseSepolia } from "wagmi/chains";
import { QueryClientProvider, QueryClient } from "@tanstack/react-query";
import { ReactNode } from "react";

// 1. Create a QueryClient instance
const queryClient = new QueryClient();

// 2. Create the Wagmi config using RainbowKit's getDefaultConfig
const config = getDefaultConfig({
  appName: "Tachi Publisher App",
  projectId: process.env.NEXT_PUBLIC_WALLETCONNECT_PROJECT_ID |

| "",
  chains:,
  ssr: true, // Required for Next.js App Router
});

// 3. Create the provider component
export function Web3Providers({ children }: { children: ReactNode }) {
  return (
    <WagmiProvider config={config}>
      <QueryClientProvider client={queryClient}>
        <RainbowKitProvider>{children}</RainbowKitProvider>
      </QueryClientProvider>
    </WagmiProvider>
  );
}
```

This Web3Providers component should then be used to wrap the children in the root layout file, src/app/layout.tsx, to make the Web3 context available throughout the entire application.

Section 8: The Publisher Onboarding Wizard

The OnboardingWizard is the centerpiece of the publisher-facing MVP. It guides the user through the essential steps of configuring and creating their on-chain license.[1] A robust and intuitive user experience is critical for building trust and encouraging adoption.

To achieve this, the wizard will combine the strengths of several libraries. Shadcn/UI's Tabs and Card components will provide the visual structure for the multi-step flow. For form state management, react-hook-form is the ideal choice due to its performance and uncontrolled component approach, which minimizes re-renders. It will be paired with zod for declarative, schema-based validation, ensuring that all user input is correct and well-formed before any on-chain transaction is initiated. This combination creates a clean separation between UI, state management, and validation logic, leading to more maintainable and reliable code.

AI Coding Agent Prompt

Code snippet

```
Generate the code for a multi-step onboarding wizard as a React client component named
`OnboardingWizard.tsx`. Use Shadcn/UI components, `react-hook-form`, and `zod`.

The component should:
1.  Be a client component (`"use client";`).
2.  Use the `Tabs` component from Shadcn/UI to manage the steps: "Site Information", "Set
Pricing", "Create License", "Deploy Gateway".
3.  Define a `zod` schema for the form, requiring a `siteName` (string, min 3 chars), `siteUrl`
(valid URL string), and `price` (a string that can be coerced to a positive number).
4.  Use `react-hook-form` with the `zodResolver` to manage and validate the form state.
5.  The "Site Information" and "Set Pricing" tabs should contain the respective form fields
(`Input` components) and display validation errors.
6.  The "Create License" tab should contain a submit `Button`. The form's `onSubmit` handler
should, for now, just log the validated form data to the console.
7.  The "Deploy Gateway" tab should display a pre-formatted code block with a placeholder
for the Cloudflare Worker script and a "Copy" button.
```

Generated Wizard Component: src/components/OnboardingWizard.tsx

TypeScript

```typescript
"use client";

import { zodResolver } from "@hookform/resolvers/zod";
import { useForm } from "react-hook-form";
import * as z from "zod";

import { Button } from "@/components/ui/button";
import {
  Form,
  FormControl,
  FormField,
  FormItem,
  FormLabel,
  FormMessage,
} from "@/components/ui/form";
import { Input } from "@/components/ui/input";
import { Tabs, TabsContent, TabsList, TabsTrigger } from "@/components/ui/tabs";
import { Card, CardContent, CardHeader, CardTitle } from "@/components/ui/card";
import { toast } from "@/components/ui/use-toast";

const formSchema = z.object({
  siteName: z.string().min(3, {
    message: "Site name must be at least 3 characters.",
  }),
  siteUrl: z.string().url({ message: "Please enter a valid URL." }),
  price: z.coerce.number().positive({ message: "Price must be a positive
number."}).transform(val => val.toString()),
});

const gatewayScriptPlaceholder = `
// This is a placeholder for your Cloudflare Worker script.
// It will be populated with your unique contract addresses
// after you create your license.
export default {
  async fetch(request, env, ctx) {
    return new Response('Hello Tachi Protocol!');
  },
};
`;

export function OnboardingWizard() {
  const form = useForm<z.infer<typeof formSchema>>({
    resolver: zodResolver(formSchema),
    defaultValues: {
      siteName: "",
      siteUrl: "",
      price: "0.005",
    },
  });

  function onSubmit(values: z.infer<typeof formSchema>) {
    // This is where the on-chain minting logic will be triggered.
    console.log("Form submitted with values:", values);
    toast({
      title: "Form Data Validated",
      description: <pre className="mt-2 w-[340px] rounded-md bg-slate-950 p-4"><code
className="text-white">{JSON.stringify(values, null, 2)}</code></pre>,
    });
  }

  return (
    <Card className="w-[600px]">
      <CardHeader>
        <CardTitle>Onboard Your Site to Tachi</CardTitle>
      </CardHeader>
      <CardContent>
        <Tabs defaultValue="site-info" className="w-full">
          <TabsList className="grid w-full grid-cols-4">
            <TabsTrigger value="site-info">1. Site Info</TabsTrigger>
            <TabsTrigger value="pricing">2. Set Price</TabsTrigger>
            <TabsTrigger value="create-license">3. Create License</TabsTrigger>
            <TabsTrigger value="deploy-gateway">4. Deploy</TabsTrigger>
          </TabsList>

          <Form {...form}>
            <form onSubmit={form.handleSubmit(onSubmit)}>
              <TabsContent value="site-info" className="pt-4">
                <div className="space-y-4">
                  <FormField
                    control={form.control}
                    name="siteName"
                    render={({ field }) => (
                      <FormItem>
                        <FormLabel>Site Name</FormLabel>
                        <FormControl>
                          <Input placeholder="My Awesome Blog" {...field} />
                        </FormControl>
                        <FormMessage />
                      </FormItem>
                    )}
                  />
                  <FormField
                    control={form.control}
                    name="siteUrl"
                    render={({ field }) => (
                      <FormItem>
                        <FormLabel>Site URL</FormLabel>
                        <FormControl>
                          <Input placeholder="https://example.com" {...field} />
                        </FormControl>
                        <FormMessage />
                      </FormItem>
                    )}
                  />
                </div>
              </TabsContent>

              <TabsContent value="pricing" className="pt-4">
                <FormField
                  control={form.control}
                  name="price"
                  render={({ field }) => (
                    <FormItem>
                      <FormLabel>Price Per Crawl (USDC)</FormLabel>
                      <FormControl>
                        <Input type="number" step="0.001" {...field} />
                      </FormControl>
                      <FormMessage />
                    </FormItem>
                  )}
                />
              </TabsContent>

              <TabsContent value="create-license" className="pt-4">
                <div className="space-y-4">
                    <p className="text-sm text-muted-foreground">
                        Review your information. By clicking the button below, you will mint your unique,
non-transferable CrawlNFT, which acts as the on-chain representation of your site's license.
                    </p>
                    <Button type="submit">Create License NFT</Button>
                </div>
              </TabsContent>
            </form>
          </Form>

          <TabsContent value="deploy-gateway" className="pt-4">
            <div className="space-y-4">
                <p className="text-sm text-muted-foreground">
                    Copy the Cloudflare Worker script below and deploy it to your site to begin
enforcing the paywall.
```

```
          </p>
          <pre className="mt-2 w-full overflow-x-auto rounded-md bg-slate-950 p-4">
            <code className="text-white">{gatewayScriptPlaceholder}</code>
          </pre>
          <Button
            onClick={() => {
              navigator.clipboard.writeText(gatewayScriptPlaceholder);
              toast({ title: "Script copied to clipboard!" });
            }}
          >
            Copy Script
          </Button>
        </div>
       </TabsContent>
      </Tabs>
     </CardContent>
   </Card>
 );
}
```

Section 9: On-Chain Interaction: Minting the License

This section implements the core Web3 functionality of the application: minting the CrawlNFT. A key architectural pattern for building clean and maintainable Web3 applications is to encapsulate complex blockchain logic within custom React hooks. This separates the concerns of the UI from the intricacies of blockchain interaction.
Instead of embedding useWriteContract and useWaitForTransactionReceipt calls directly within the OnboardingWizard component, a custom hook, useMintCrawlNFT, will be created. This hook will manage the entire lifecycle of the minting transaction—preparation, sending, and waiting for confirmation. The UI component will then consume this hook, receiving a simple API (mint function) and state flags (isMinting, isSuccess). This approach makes the UI component significantly cleaner, more declarative, and easier to reason about, while the complex Web3 logic is neatly contained and reusable.[22]

AI Coding Agent Prompts

Custom Minting Hook (useMintCrawlNFT.ts)

Code snippet

Generate a custom React hook `useMintCrawlNFT.ts` for a Next.js app using `wagmi`.

The hook should:
1. Be in its own file: `src/hooks/useMintCrawlNFT.ts`.
2. Import the ABI and contract address for the `CrawlNFT` contract from a configuration file.
3. Use `wagmi`'s `useWriteContract` hook to prepare a transaction for the `mint` function of the contract.
4. Use `wagmi`'s `useWaitForTransactionReceipt` hook to track the confirmation status of the transaction hash returned by `useWriteContract`.
5. The hook should accept an `onSuccess` callback function as an argument.
6. Use a `useEffect` to call the `onSuccess` callback when the transaction is successfully confirmed.
7. Return a consolidated state object including:
   - A `mint` function that takes the `publisherAddress` and `termsURI` as arguments and calls `writeContract`.
   - An `isMinting` boolean that is `true` if either the write transaction is pending or the receipt is loading.
   - An `isSuccess` boolean that is `true` only after the transaction is confirmed.
   - The `hash` of the transaction.
   - Any `error` object.

Integration into OnboardingWizard.tsx

Code snippet

Show how to integrate the `useMintCrawlNFT` custom hook into the `OnboardingWizard.tsx` component.

The updated component should:
1. Import and call the `useMintCrawlNFT` hook.
2. In the `onSubmit` function, instead of logging to the console, it should now call the `mint` function from the hook, passing the connected user's address and a placeholder IPFS hash for the terms.
3. The "Create License" button's text and `disabled` state should react to the `isMinting` status from the hook.

4. The `onSuccess` callback passed to the hook should display a success `Toast` and programmatically switch the `Tabs` component to the "Deploy Gateway" step.

Generated Code

src/config/contracts.ts (New File)

This file will store the contract addresses and ABIs, keeping them organized and separate from the component logic.

TypeScript

```
// For demonstration, the ABI is simplified. In a real project, import it from the artifacts.
export const CRAWL_NFT_ABI =,
   "name": "mint",
   "outputs":,
   "stateMutability": "nonpayable",
   "type": "function"
  }
] as const;

export const CRAWL_NFT_ADDRESS = "0xYourDeployedCrawlNftAddress"; // Replace with your deployed contract address
```

src/hooks/useMintCrawlNFT.ts (New File)

TypeScript

```
"use client";

import { useEffect } from "react";
import { type BaseError, useWriteContract, useWaitForTransactionReceipt } from "wagmi";
import { CRAWL_NFT_ABI, CRAWL_NFT_ADDRESS } from "@/config/contracts";
import { toast } from "@/components/ui/use-toast";
```

```
interface UseMintCrawlNFTProps {
  onSuccess?: (hash: `0x${string}`) => void;
}

export function useMintCrawlNFT({ onSuccess }: UseMintCrawlNFTProps = {}) {
  const { data: hash, error, isPending, writeContract } = useWriteContract();

  const { isLoading: isConfirming, isSuccess: isConfirmed } =
    useWaitForTransactionReceipt({
      hash,
    });

  useEffect(() => {
    if (isConfirmed && hash) {
      toast({
        title: "License Minted Successfully!",
        description: `Transaction confirmed. Hash: ${hash}`,
      });
      onSuccess?.(hash);
    }
    if (error) {
        toast({
            title: "Minting Failed",
            description: (error as BaseError)?.shortMessage |

| error.message,
            variant: "destructive",
        });
    }
  },);

  const mint = (publisherAddress: `0x${string}`, termsURI: string) => {
    writeContract({
      address: CRAWL_NFT_ADDRESS,
      abi: CRAWL_NFT_ABI,
      functionName: "mint",
      args:,
    });
  };

  return {
    mint,
    isMinting: isPending |

| isConfirming,
    isSuccess: isConfirmed,
    hash,
    error,
  };
}
```

src/components/OnboardingWizard.tsx (Updated)

This shows the integration of the custom hook into the form's submission logic.

TypeScript

```
//... (imports from previous OnboardingWizard)
import { useState } from "react";
import { useAccount } from "wagmi";
import { useMintCrawlNFT } from "@/hooks/useMintCrawlNFT";

//... (formSchema and gatewayScriptPlaceholder remain the same)

export function OnboardingWizard() {
  const = useState("site-info");
  const { address: connectedAddress } = useAccount();

  const { mint, isMinting, isSuccess } = useMintCrawlNFT({
    onSuccess: () => {
      // On successful mint, move to the final step
      setActiveTab("deploy-gateway");
    },
  });

  const form = useForm<z.infer<typeof formSchema>>({
    //... (form setup remains the same)
  });

  function onSubmit(values: z.infer<typeof formSchema>) {
    if (!connectedAddress) {
      toast({ title: "Please connect your wallet first.", variant: "destructive" });
      return;
    }
    // In a real app, the terms would be uploaded to IPFS first to get a real hash.
    const placeholderTermsURI =
"ipfs://bafybeigdyrzt5sfp7udm7hu76uh7y26nf3efuylqabf3oclgtqy55fbzdi";

    mint(connectedAddress, placeholderTermsURI);
  }

  return (
    <Card className="w-[600px]">
      {/*... (CardHeader remains the same) */}
      <CardContent>
        <Tabs value={activeTab} onValueChange={setActiveTab} className="w-full">
          {/*... (TabsList remains the same) */}

          <Form {...form}>
            <form onSubmit={form.handleSubmit(onSubmit)}>
              {/*... (TabsContent for site-info and pricing remain the same) */}

              <TabsContent value="create-license" className="pt-4">
                <div className="space-y-4">
                    <p className="text-sm text-muted-foreground">
                        Review your information. By clicking the button below, you will mint your unique,
non-transferable CrawlNFT.
                    </p>
                    <Button type="submit" disabled={isMinting}>
                        {isMinting? "Minting..." : "Create License NFT"}
                    </Button>
                </div>
              </TabsContent>
            </form>
          </Form>

          {/*... (TabsContent for deploy-gateway remains the same) */}
        </Tabs>
      </CardContent>
    </Card>
  );
}
```

Section 10: The Publisher Dashboard

The dashboard is where the publisher sees the tangible results of participating in the protocol. It must display verifiable, on-chain data to build trust and demonstrate value. A purely client-side approach to fetching this data, while possible, has significant security and performance drawbacks. It would expose the RPC provider's endpoint and API key in the browser and could lead to inefficient data fetching.
The superior architectural pattern is to use a Next.js API Route as a Backend-for-Frontend (BFF).[24] This creates a secure, server-side intermediary that handles the direct communication with the blockchain. The front-end makes a simple, authenticated request to this API route, which then uses a
viem public client to securely query the ProofOfCrawlLedger contract for CrawlLogged events. This BFF pattern enhances security by hiding credentials, improves performance by allowing for server-side data aggregation and caching, and provides a more scalable architecture for the application.[1]

AI Coding Agent Prompts

API Route (src/app/api/crawls/route.ts)

Code snippet

Generate the code for a Next.js API route in `src/app/api/crawls/route.ts`.

1. Create a `GET` handler.
2. The handler should read a `publisherAddress` from the URL's search parameters. If it's missing or invalid, return a 400 error.
3. Inside the handler, create a `viem` `publicClient` configured for Base Sepolia using `createPublicClient` and `http`.
4. Use `publicClient.getLogs` to query for `CrawlLogged` events from the `ProofOfCrawlLedger` contract.
5. The filter should specify the contract address (from an environment variable), the event signature, and the indexed `publisherWallet` topic to filter for events related to the specific publisher.
6. The `CrawlLogged` event has the signature: `event CrawlLogged(address indexed crawlerWallet, address indexed publisherWallet, bytes32 contentHash, uint256 timestamp)`.
7. Parse the retrieved logs into a clean JSON array containing `crawler`, `contentHash`, and

`timestamp` for each event, and return it with a 200 status code. Include robust error handling.

Dashboard Component (src/components/DashboardView.tsx)

Code snippet

Generate the code for a React client component `DashboardView.tsx` that displays the crawl history.

The component should:
1. Use a data-fetching hook (like `swr` or `react-query`'s `useQuery`) to fetch data from the `/api/crawls?publisherAddress=<address>` endpoint. You will need to get the connected user's address from `wagmi`'s `useAccount` hook.
2. Handle loading and error states gracefully.
3. Use Shadcn/UI's `Table`, `TableHeader`, `TableRow`, `TableHead`, `TableBody`, and `TableCell` to render the returned crawl logs in a structured table with columns for "Crawler Address", "Content Hash", and "Timestamp".

Generated Code

src/app/api/crawls/route.ts (New File)

TypeScript

```
import { type NextRequest, NextResponse } from "next/server";
import { createPublicClient, http, parseAbiItem, decodeEventLog } from "viem";
import { baseSepolia } from "viem/chains";

// This ABI item is crucial for Viem to understand the event structure
const CRAWL_LOGGED_EVENT_ABI = parseAbiItem(
  "event CrawlLogged(address indexed crawlerWallet, address indexed publisherWallet,
bytes32 contentHash, uint256 timestamp)"
```

```
);

const PROOF_OF_CRAWL_LEDGER_ADDRESS =
process.env.PROOF_OF_CRAWL_LEDGER_ADDRESS as `0x${string}`;
const BASE_SEPOLIA_RPC_URL = process.env.BASE_SEPOLIA_RPC_URL;

const publicClient = createPublicClient({
  chain: baseSepolia,
  transport: http(BASE_SEPOLIA_RPC_URL),
});

export async function GET(request: NextRequest) {
  const { searchParams } = new URL(request.url);
  const publisherAddress = searchParams.get("publisherAddress") as `0x${string}` | null;

  if (!publisherAddress) {
    return NextResponse.json({ error: "Publisher address is required" }, { status: 400 });
  }

  try {
    const logs = await publicClient.getLogs({
      address: PROOF_OF_CRAWL_LEDGER_ADDRESS,
      event: CRAWL_LOGGED_EVENT_ABI,
      args: {
        // This filters by the indexed publisherWallet parameter
        publisherWallet: publisherAddress,
      },
      fromBlock: "earliest",
      toBlock: "latest",
    });

    const parsedLogs = logs.map((log) => ({
      crawler: log.args.crawlerWallet,
      contentHash: log.args.contentHash,
      timestamp: log.args.timestamp?.toString(),
      transactionHash: log.transactionHash,
    }));

    return NextResponse.json(parsedLogs);
  } catch (error) {
    console.error("Failed to fetch crawl logs:", error);
    return NextResponse.json({ error: "Failed to fetch crawl logs" }, { status: 500 });
  }
```

```
}
```

src/components/DashboardView.tsx (New File)

TypeScript

```typescript
"use client";

import { useAccount } from "wagmi";
import useSWR from "swr";
import {
  Table,
  TableBody,
  TableCell,
  TableHead,
  TableHeader,
  TableRow,
} from "@/components/ui/table";
import { Card, CardContent, CardHeader, CardTitle } from "@/components/ui/card";

interface CrawlLog {
  crawler: string;
  contentHash: string;
  timestamp: string;
  transactionHash: string;
}

const fetcher = (url: string) => fetch(url).then((res) => res.json());

export function DashboardView() {
  const { address, isConnected } = useAccount();

  const { data: crawlLogs, error, isLoading } = useSWR<CrawlLog>(
    isConnected && address? `/api/crawls?publisherAddress=${address}` : null,
    fetcher,
    { refreshInterval: 30000 } // Poll for new logs every 30 seconds
  );

  if (!isConnected) {
```

```
  return <p>Please connect your wallet to view your dashboard.</p>;
}

if (isLoading) {
  return <p>Loading crawl history...</p>;
}

if (error) {
  return <p>Failed to load data. Please try again later.</p>;
}

return (
  <Card className="w-full">
    <CardHeader>
      <CardTitle>Recent Crawl Activity</CardTitle>
    </CardHeader>
    <CardContent>
      <Table>
        <TableHeader>
          <TableRow>
            <TableHead>Crawler Address</TableHead>
            <TableHead>Timestamp</TableHead>
            <TableHead>Transaction</TableHead>
          </TableRow>
        </TableHeader>
        <TableBody>
          {crawlLogs && crawlLogs.length > 0? (
            crawlLogs.map((log) => (
              <TableRow key={log.transactionHash}>
                <TableCell className="font-mono text-xs">{log.crawler}</TableCell>
                <TableCell>{new Date(Number(log.timestamp) * 1000).toLocaleString()}</TableCell>
                <TableCell>
                  <a
                    href={`https://sepolia.basescan.org/tx/${log.transactionHash}`}
                    target="_blank"
                    rel="noopener noreferrer"
                    className="text-blue-500 hover:underline"
                  >
                    View on Basescan
                  </a>
                </TableCell>
              </TableRow>
            ))
          ) : (
            <TableRow>
              <TableCell colSpan={3} className="text-center">
                No crawl activity recorded yet.
              </TableCell>
            </TableRow>
          )}
        </TableBody>
      </Table>
    </CardContent>
  </Card>
 );
}
```

Part III: The Protocol's Enforcement and Client Layers

This part covers the critical off-chain components that connect the on-chain rules to the real-world web, enforcing the paywall and enabling AI crawlers to participate in the ecosystem. These components are the active agents that translate the passive logic of the smart contracts into live, dynamic interactions.

Section 11: The Publisher Gateway: A Cloudflare Worker

The Publisher Gateway is the protocol's real-time enforcement arm. Implemented as a serverless Cloudflare Worker, it runs at the edge, intercepting every request to a publisher's site before it hits the origin server.[1] This allows it to act as a trustless executor for the on-chain rules.
The worker's logic is the critical bridge between the declarative rules on the blockchain and the imperative actions of the web. It does not blindly trust a crawler's claim of payment; it actively *verifies* the payment by making a direct, secure RPC call to a Base node.[26] This on-chain verification is the core of the system's security. The performance of this worker is paramount, as any latency in the verification process directly impacts the "time to first byte" for the paying crawler. To mitigate this, the worker uses the
ctx.waitUntil() method, a feature of the Cloudflare Workers runtime, to perform the final on-chain logging action asynchronously, after the content has already been sent to the crawler. This ensures the audit trail is complete without penalizing the crawler's performance.[1]

AI Coding Agent Prompt

Code snippet

Generate the complete TypeScript code for a Cloudflare Worker (`src/index.ts`) that implements the Tachi Pay-Per-Crawl gateway.

The worker should:
1. Define an `Env` interface for secrets and environment variables: `PAYMENT_PROCESSOR_ADDRESS`, `CRAWL_PRICE_USDC`, `BASE_RPC_URL`, and `PROOF_OF_CRAWL_LEDGER_ADDRESS`.
2. Import `createPublicClient`, `http`, `parseAbiItem`, and `decodeEventLog` from `viem`.
3. In the `fetch` handler, maintain a list of known AI bot `User-Agent` strings.
4. If an incoming request's `User-Agent` matches a known bot:
    a. Check for an `Authorization: Bearer <tx_hash>` header.
    b. If no header is found, return an `HTTP 402 Payment Required` response. This response must include custom `x402-` headers containing the `price`, `recipient` (the payment processor address), `token` (USDC), and `network` (base), all sourced from the `env`.
    c. If an `Authorization` header is present, extract the transaction hash.
    d. Create a `viem` `publicClient` using the `BASE_RPC_URL` from the environment.
    e. Call `publicClient.getTransactionReceipt` to fetch the receipt for the provided hash.
    f. Parse the receipt's logs to find the USDC `Transfer` event. Verify that the `to` address matches the `PAYMENT_PROCESSOR_ADDRESS` and the `value` is sufficient.
    g. If verification is successful, `fetch` the original request to the origin server to get the content. Asynchronously, using `ctx.waitUntil()`, prepare and send a transaction to call the `logCrawl` function on the `ProofOfCrawlLedger` contract.
    h. If verification fails for any reason, return a `403 Forbidden` response.
5. If the request is not from a known bot, pass it through to the origin server untouched using `fetch(request)`.

Generated Cloudflare Worker: src/index.ts

TypeScript

```
import { createPublicClient, http, getContract, createWalletClient, keccak256, toHex } from
'viem';
import { privateKeyToAccount } from 'viem/accounts';
import { baseSepolia } from 'viem/chains';
```

```
// Define the environment variables and secrets expected in wrangler.toml
export interface Env {
  PAYMENT_PROCESSOR_ADDRESS: `0x${string}`;
  CRAWL_PRICE_USDC: string; // e.g., "5000" for 0.005 USDC (6 decimals)
  BASE_RPC_URL: string;
  PROOF_OF_CRAWL_LEDGER_ADDRESS: `0x${string}`;
  GATEWAY_SIGNER_PRIVATE_KEY: string; // The private key for the worker to call logCrawl
  USDC_TOKEN_ADDRESS: `0x${string}`;
}

// ABIs for viem to interact with contracts
const usdcAbi =;
const proofOfCrawlLedgerAbi =;

// List of known AI crawlers to challenge
const AI_CRAWLER_USER_AGENTS =;

export default {
  async fetch(request: Request, env: Env, ctx: ExecutionContext): Promise<Response> {
    const userAgent = request.headers.get('User-Agent') |

| '';
    const isKnownAICrawler = AI_CRAWLER_USER_AGENTS.some(ua =>
userAgent.includes(ua));

    if (!isKnownAICrawler) {
      // Not a target bot, pass through to origin
      return fetch(request);
    }

    const authHeader = request.headers.get('Authorization');
    if (!authHeader ||!authHeader.startsWith('Bearer ')) {
      // It's a bot, but there's no payment proof. Issue a 402 challenge.
      return new Response(JSON.stringify({ error: 'Payment Required' }), {
        status: 402,
        statusText: 'Payment Required',
        headers: {
          'Content-Type': 'application/json',
          'x402-price': env.CRAWL_PRICE_USDC,
          'x402-recipient': env.PAYMENT_PROCESSOR_ADDRESS,
          'x402-token': env.USDC_TOKEN_ADDRESS,
          'x402-network': 'base-sepolia',
        },
      });
    }

    // It's a bot with a payment proof. Verify it.
    const txHash = authHeader.substring(7) as `0x${string}`;

    const publicClient = createPublicClient({
      chain: baseSepolia,
      transport: http(env.BASE_RPC_URL),
    });

    try {
      const receipt = await publicClient.getTransactionReceipt({ hash: txHash });

      if (receipt.status!== 'success') {
        return new Response(JSON.stringify({ error: 'Payment transaction failed' }), { status: 403
});
      }

      // Find the USDC transfer event in the logs
      const transferLog = receipt.logs.find(log => log.address.toLowerCase() ===
env.USDC_TOKEN_ADDRESS.toLowerCase());

      if (!transferLog) {
        return new Response(JSON.stringify({ error: 'No USDC transfer event found' }), { status:
403 });
      }

      const decodedEvent = decodeEventLog({
        abi: usdcAbi,
        data: transferLog.data,
        topics: transferLog.topics,
      });

      const { from: crawlerWallet, to: recipient, value: amountPaid } = decodedEvent.args;

      // Verification checks
      if (recipient?.toLowerCase()!== env.PAYMENT_PROCESSOR_ADDRESS.toLowerCase()) {
        return new Response(JSON.stringify({ error: 'Payment sent to incorrect recipient' }), {
status: 403 });
      }
      if (BigInt(amountPaid) < BigInt(env.CRAWL_PRICE_USDC)) {
        return new Response(JSON.stringify({ error: 'Insufficient payment amount' }), { status:
403 });
      }

      // Verification successful. Serve content and log the crawl asynchronously.
      const originResponse = await fetch(request);

      ctx.waitUntil(logCrawlEvent(request, env, crawlerWallet, receipt.from));

      return originResponse;

    } catch (error) {
      console.error('Verification failed:', error);
      return new Response(JSON.stringify({ error: 'Invalid payment proof' }), { status: 403 });
    }
  },
};

async function logCrawlEvent(request: Request, env: Env, crawlerWallet: `0x${string}`,
publisherWallet: `0x${string}`) {
    try {
        const account = privateKeyToAccount(env.GATEWAY_SIGNER_PRIVATE_KEY as
`0x${string}`);
        const walletClient = createWalletClient({
            account,
            chain: baseSepolia,
            transport: http(env.BASE_RPC_URL),
        });

        const urlHash = keccak256(toHex(request.url));

        await walletClient.writeContract({
            address: env.PROOF_OF_CRAWL_LEDGER_ADDRESS,
            abi: proofOfCrawlLedgerAbi,
            functionName: 'logCrawl',
            args:,
        });
        console.log(`Successfully logged crawl for ${request.url}`);
    } catch (e) {
        console.error("Failed to log crawl event:", e);
    }
}
```

Section 12: The AI Crawler SDK: A Node.js PoC

The AI Crawler SDK is the product offered to the demand side of the marketplace. Its primary goal is to abstract away the complexity of blockchain interaction, providing a familiar fetch-like interface to AI developers.[1] The key technology that enables this seamless experience is ERC-4337 Account Abstraction.[1]
An AI crawler is an autonomous agent. It cannot have a human sign every payment transaction, and storing a raw private key on a server is a severe security vulnerability. Account Abstraction solves this by giving the crawler a programmable smart contract wallet instead of a simple key-based account.[1] This allows for enhanced security, such as storing the master key in a secure vault (e.g., AWS KMS) and allowing the crawler only to request signatures, not access the key itself. It also enables gas sponsorship via a "Paymaster," allowing a central entity to cover the network fees for its fleet of crawlers.[1]
Libraries like Alchemy's aa-sdk make implementing this accessible. They provide high-level clients that handle the construction and dispatch of UserOperation objects—the core data structure of ERC-4337. The SDK's main task is to catch a 402 response, parse the payment details, construct the correct UserOperation for a USDC transfer, and send it to a Bundler for execution on the blockchain.[28]

AI Coding Agent Prompt

Code snippet

```
Generate a TypeScript function `fetchWithTachi(url: string)` for a Node.js environment that
implements the client-side logic for the Pay-Per-Crawl protocol using Alchemy's Account
Abstraction SDK (`@alchemy/aa-core` and `@alchemy/aa-alchemy`).

The function should:
1.  Take a URL string as input.
2.  Make an initial `fetch` request.
3.  If the response status is not 402, it should return the response immediately.
4.  If the status is 402:
    a. Parse the `x402-price`, `x402-recipient`, `x402-token`, and `x402-network` headers.
    b. Assume an `AlchemyProvider` (`provider`) has been initialized elsewhere with a signer
(the crawler's smart wallet owner) and an Alchemy API key.
    c. Use `viem`'s `encodeFunctionData` to construct the `calldata` for a USDC `transfer` call.
The recipient should be the `PaymentProcessor` address and the amount should be the price
```

from the headers.
    d. Use `provider.sendUserOperation` with the `target` being the USDC contract address and the `data` being the `calldata` generated in the previous step.
    e. Await the `UserOperationResponse` to get the hash and then use `provider.waitForUserOperationTransaction` to get the final transaction hash.
    f. Make a second `fetch` request to the original URL, this time adding the `Authorization: Bearer <transactionHash>` header.
    g. Return the response from the second, successful fetch.
5. Include robust error handling for failed fetches and invalid 402 responses.

Generated AI Crawler SDK Function

TypeScript

```
import { AlchemyProvider } from "@alchemy/aa-alchemy";
import { Address, encodeFunctionData, parseAbi } from "viem";

// This ABI is needed to encode the transfer call
const erc20Abi = parseAbi([
  'function transfer(address to, uint256 amount) returns (bool)',
]);

/**
 * Fetches a URL, handling the Tachi Pay-Per-Crawl 402 payment flow.
 * @param url The URL to fetch.
 * @param provider An initialized AlchemyProvider for the crawler's smart wallet.
 * @returns The final Response object after payment.
 */
export async function fetchWithTachi(
  url: string,
  provider: AlchemyProvider
): Promise<Response> {
  // 1. Make the initial request
  const initialRes = await fetch(url);

  // 2. If no payment is required, return the response
  if (initialRes.status!== 402) {
    return initialRes;
  }
```

```
console.log("Payment required. Handling 402 challenge...");

// 3. Parse payment details from headers
const price = initialRes.headers.get("x402-price");
const recipient = initialRes.headers.get("x402-recipient") as Address | null;
const token = initialRes.headers.get("x402-token") as Address | null;

if (!price ||!recipient ||!token) {
  throw new Error("Invalid 402 response: missing payment headers.");
}

// 4. Construct the UserOperation
console.log(`Constructing payment of ${price} USDC to ${recipient}`);
const calldata = encodeFunctionData({
  abi: erc20Abi,
  functionName: "transfer",
  args:,
});

try {
  // 5. Send the UserOperation via the provider
  const uoResponse = await provider.sendUserOperation({
    target: token,
    data: calldata,
    value: 0n, // We are sending ERC20 tokens, not native ETH
  });

  console.log(`UserOperation sent. Hash: ${uoResponse.hash}`);

  // 6. Wait for the transaction to be mined
  const txHash = await provider.waitForUserOperationTransaction(uoResponse);
  console.log(`Transaction confirmed. Hash: ${txHash}`);

  // 7. Make the second request with payment proof
  const finalRes = await fetch(url, {
    headers: {
      Authorization: `Bearer ${txHash}`,
    },
  });

  console.log("Successfully fetched content with payment proof.");
  return finalRes;
  } catch (error) {
    console.error("Failed to process payment:", error);
    throw new Error("Payment execution failed.");
  }
}
```

Part IV: Integration and End-to-End Validation

This final part brings all the previously developed components together—the on-chain contracts, the publisher front-end, the gateway worker, and the crawler SDK—to ensure they function as a single, cohesive system. A successful end-to-end test serves as the ultimate validation of the protocol's architecture and is a critical business milestone, moving the project from a collection of disparate parts to a tangible, demonstrable product.

Section 13: The "Happy Path": A Full System Test

This section provides a clear, step-by-step instructional walkthrough for conducting a full end-to-end test of the deployed MVP. It is designed to be followed sequentially by the developer to confirm that every component is communicating correctly. A successful run of this "happy path" scenario is the most powerful tool for showcasing the protocol's functionality to potential partners and investors.[1]

Test Scenario Walkthrough

Step 1: Prerequisites and Setup

Before beginning the test, ensure the following conditions are met:

- **Contracts Deployed:** The CrawlNFT, PaymentProcessor, and ProofOfCrawlLedger smart contracts have been successfully deployed to the Base Sepolia testnet using the Hardhat script from Part I, Section 5. The deployed addresses have been recorded.
- **Environment Variables:** All necessary .env files and Cloudflare Worker secrets are populated with the correct deployed contract addresses, a Base Sepolia RPC URL, and the required private keys.
- **Publisher App Running:** The tachi-publisher-app Next.js application is running locally (pnpm dev) or has been deployed to a preview environment like Vercel.

- **Funded Wallets:**
  - The **Publisher Wallet** (the one you will connect to the app) is funded with a small amount of Base Sepolia ETH for gas fees. This can be obtained from a faucet listed in the reference table.
  - The **Crawler's Smart Wallet** (the one whose owner's private key is used by the SDK) is funded with testnet USDC. This can be obtained from Circle's public faucet by sending funds to the smart wallet's address.[18]

Step 2: Publisher Onboarding

1. Navigate to the running Tachi Publisher App in your browser.
2. Connect your **Publisher Wallet** using the RainbowKit "Connect Wallet" button. Ensure you are connected to the Base Sepolia network.
3. Using the OnboardingWizard component, fill in your test site's information (e.g., Name: "My Test Site", URL: https://my-test-site.pages.dev).
4. Set a price (e.g., 0.001 USDC).
5. Proceed to the "Create License" tab and click the "Create License NFT" button.
6. Your wallet (e.g., MetaMask) will prompt you to sign and send the minting transaction. Confirm it.
7. Wait for the transaction to be confirmed. The UI should provide feedback, and upon success, you should see a confirmation toast and be advanced to the final step.

Step 3: Gateway Deployment

1. In the final "Deploy Gateway" step of the wizard, a placeholder for the Cloudflare Worker script is displayed. In a production version, this would be dynamically populated with the newly minted license details. For this test, manually copy the src/index.ts worker code from Part III, Section 11.
2. Go to your Cloudflare dashboard and create a new Worker.
3. Paste the worker code into the editor.
4. Navigate to the worker's settings and add the required environment variables and secrets (PAYMENT_PROCESSOR_ADDRESS, CRAWL_PRICE_USDC, etc.) with the values from your deployment.
5. Add a trigger (a route) for the worker, pointing it to your test website (e.g., my-test-site.pages.dev/*).
6. Deploy the worker.

Step 4: AI Crawler Execution

1. Open the Node.js project containing the fetchWithTachi SDK function from Part III,

Section 12.
2. Ensure the script is configured with an initialized AlchemyProvider that uses the signer for your funded **Crawler's Smart Wallet**.
3. Execute the script, calling fetchWithTachi with a URL from your test site (e.g., https://my-test-site.pages.dev/article/1).

Step 5: Comprehensive Verification

A successful test is confirmed by observing the correct behavior at every layer of the protocol:

1. **Crawler Console:** The terminal running the crawler script should first log "Payment required. Handling 402 challenge...", followed by the UserOperation hash, the confirmed transaction hash, and finally, "Successfully fetched content with payment proof." The script should exit without errors.
2. **Gateway Logs:** In the Cloudflare dashboard, view the live logs for your deployed worker. You should see an initial request from the crawler that results in a 402 response. This should be followed by a second request containing the Authorization header, which the worker successfully verifies. Finally, you should see the log message "Successfully logged crawl for...".
3. **Block Explorer (Basescan):**
    - Take the transaction hash logged by the crawler or the gateway and search for it on Base Sepolia Basescan (https://sepolia.basescan.org).
    - On the transaction details page, confirm that a USDC Transfer occurred from the crawler's smart wallet to the PaymentProcessor contract for the correct amount.
    - Check the "Logs" tab of the transaction for the CrawlLogged event emitted by the ProofOfCrawlLedger contract. The event data should show the correct crawler and publisher wallet addresses.
4. **Publisher Dashboard:**
    - Return to the Tachi Publisher App.
    - Navigate to the dashboard view.
    - The "Recent Crawl Activity" table should now display a new row corresponding to the test crawl you just executed, with the correct crawler address and timestamp, and a link to the transaction on Basescan.

Successfully completing this entire sequence validates that all components are integrated correctly and that the core economic and technical loop of the Tachi protocol is fully functional.

Works cited

1. Pay-Per-Crawl Startup Technical Plan
2. Install and Setup Foundry for Solidity - Ethereum Blockchain Developer, accessed July 16, 2025, https://www.ethereum-blockchain-developer.com/courses/ethereum-course-202

[4/project-erc721-nft-with-remix-truffle-hardhat-and-foundry/install-and-configure-foundry-forge]()
3. Integrating with Foundry | Ethereum development environment for professionals by Nomic Foundation - Hardhat, accessed July 16, 2025, https://hardhat.org/hardhat-runner/docs/advanced/hardhat-and-foundry
4. 7. Deploying to a live network | Ethereum development environment for professionals by Nomic Foundation - Hardhat, accessed July 16, 2025, https://hardhat.org/tutorial/deploying-to-a-live-network
5. Deploying Smart Contracts - Base Documentation, accessed July 16, 2025, https://docs.base.org/learn/hardhat/hardhat-deploy/hardhat-deploy-sbs
6. foundry.toml - pcaversaccio/hardhat-project-template-ts - GitHub, accessed July 16, 2025, https://github.com/pcaversaccio/hardhat-project-template-ts/blob/main/foundry.toml
7. Secure Ether Transfer | solidity-patterns - GitHub Pages, accessed July 16, 2025, https://fravoll.github.io/solidity-patterns/secure_ether_transfer.html
8. Smart Contract Security | By RareSkills, accessed July 16, 2025, https://rareskills.io/post/smart-contract-security
9. @openzeppelin/contracts - npm, accessed July 16, 2025, https://www.npmjs.com/package/@openzeppelin/contracts
10. Payment - OpenZeppelin Docs, accessed July 16, 2025, https://docs.openzeppelin.com/contracts/2.x/api/payment
11. Ethereum Sepolia Faucet - Google Cloud, accessed July 16, 2025, https://cloud.google.com/application/web3/faucet/ethereum/sepolia
12. Testnet Faucets - Circle Docs, accessed July 16, 2025, https://developers.circle.com/w3s/developer-console-faucet
13. USDC Contract Addresses - Circle Docs, accessed July 16, 2025, https://developers.circle.com/stablecoins/usdc-contract-addresses
14. USDC - CCIP Supported Tokens - Chainlink Documentation, accessed July 16, 2025, https://docs.chain.link/ccip/directory/testnet/token/USDC
15. USDC Support - Crossmint Docs, accessed July 16, 2025, https://docs.crossmint.com/payments/advanced/usdc-support
16. Base: Deploy an ERC-721 contract with Hardhat - Chainstack Docs, accessed July 16, 2025, https://docs.chainstack.com/docs/base-tutorial-deploy-an-erc-721-contract-with-hardhat
17. Network Faucets - Base Documentation, accessed July 16, 2025, https://docs.base.org/base-chain/tools/network-faucets
18. Testnet Faucet | Circle, accessed July 16, 2025, https://faucet.circle.com/
19. Next.js - Shadcn UI, accessed July 16, 2025, https://ui.shadcn.com/docs/installation/next
20. Installation - RainbowKit, accessed July 16, 2025, https://rainbowkit.com/docs/installation
21. Migration Guide - RainbowKit, accessed July 16, 2025, https://rainbowkit.com/docs/migration-guide

22. Write to Contract | Wagmi, accessed July 16, 2025, https://wagmi.sh/react/guides/write-to-contract
23. useWriteContract - Wagmi, accessed July 16, 2025, https://wagmi.sh/react/api/hooks/useWriteContract
24. Next.js API Routes Building Server-Side Functionality - DEV Community, accessed July 16, 2025, https://dev.to/tianyaschool/nextjs-api-routes-building-server-side-functionality-2fg6
25. Routing: API Routes - Next.js, accessed July 16, 2025, https://nextjs.org/docs/pages/building-your-application/routing/api-routes
26. HTTP Transport - Viem, accessed July 16, 2025, https://viem.sh/docs/clients/transports/http.html
27. Fetch · Cloudflare Workers docs, accessed July 16, 2025, https://developers.cloudflare.com/workers/runtime-apis/fetch/
28. Build Account Abstraction Wallets with Alchemy and AWS: Part 2, accessed July 16, 2025, https://aws.amazon.com/blogs/web3/build-account-abstraction-wallets-with-alchemy-and-aws-part-2/
29. Learn How to Use the AA-SDK | Account Abstraction | Solidity | Viem - YouTube, accessed July 16, 2025, https://www.youtube.com/watch?v=7aJ5bonMSyQ