PATCHWORK

Whitepaper

Contents

Introduction 1

- Background on EVM App Development 1
- The Patchwork Vision 1

Challenges in EVM App Development 3

- App Development Complexities 3
- High Gas Fees from Inefficient Storage 3
- Challenges in Data Provenance and Ownership Tracking 4
- Lack of Seamless Onchain App Interoperability 5

Patchwork's Solution 6

Core Concepts 8

- The Patchwork721 Standard 8
- Patches and Fragments 8
- Assignees and Scopes 9
- Locks and Freezes 10

The Patchwork Stack: Simplifying App Development 11

- Base Contracts 12
- Data Modeling: Metadata Standard 12
- Defining Relationships: Patchwork Protocol 13
- Developer Tooling 14

Example Use Cases 18

- Composable Game Items 18
- Onchain Transactions & Rewards 18
- Verified Trust Systems 19
- Soulbound Tokens 19

Roadmap 20

Conclusion 21

- Get Started With Patchwork 21
- Who's Behind Patchwork 21

Introduction

Background on EVM App Development

Building applications (apps) on the Ethereum Virtual Machine (EVM), such as those on networks like Polygon or Base, requires navigating the complexities of the EVM, a decentralized computing environment that processes smart contracts. The EVM's stack-based architecture presents significant challenges for developers, easily leading to inefficiencies, possibly high gas fees, and lack of data provenance or interoperability if not carefully managed.

Despite numerous Ethereum Improvement Proposals (EIPs) that attempt to tackle these issues, many lack cohesion, making it difficult for emerging apps to integrate these solutions seamlessly. For example, the ERC-721 contract facilitates unique asset creation (NFTs), but falls short in supporting complex asset relationships, such as associating a governance token with multiple underlying voting rights or assets. This pushes developers to adopt inefficient workarounds or abandon their ideas altogether.

The Patchwork Vision

Since the early days of Web3, we've been deeply immersed in building and innovating, witnessing firsthand the challenges of developing on EVM, and over time, noticed we weren't alone. Many other developers abandoned projects or turned to Web2 solutions due to overwhelming complexity and fragmented tools. This drove us to create Patchwork – transforming the experience of building Web3 apps as simple as describing an idea.

Imagine explaining your app and having a solution seamlessly generate interoperable smart contracts

and backend infrastructure with data entirely onchain. Recent advancements in Large Language Models (LLMs) have made this kind of seamless development feel more attainable than ever. However, while LLMs can accelerate coding, smart contract developers know that onchain apps demand absolute security, with no room for unsafe or inefficient code.

Patchwork embraces automation while prioritizing security, enabling rapid Web3 app development without compromising reliability. By extending ERC-721 into a new realm of dynamic, standardized assets, the Patchwork stack empowers people to safely streamline workflows, cutting deployment time from months to just hours. From crypto enthusiasts building quick, disposable mini-apps to developers seeking a comprehensive full-stack solution for ambitious projects, Patchwork provides the tools and flexibility to meet a wide variety of use cases.

Challenges in EVM App Development

App Development Complexities

A key component of building EVM-based apps is creating smart contracts. Despite EVM's popularity, the smart contract building process is deceptively complex[1]. Developers must balance gas efficiency with functionality, as every operation consumes gas, which directly translates to costs. Poor optimization can render a contract impractical or even fail to deploy.

Mistakes in design can lead to security vulnerabilities, such as reentrancy attacks or unintentional exploits, making rigorous auditing essential. For instance, a project called Wolf Game[2] had to rebuild its entire ecosystem after discovering a bug in its immutable smart contract. These challenges are further compounded by the lack of robust tools for managing inter-contract relationships and metadata, often leaving developers to devise makeshift solutions. As a result, even seemingly simple apps can face significant hurdles in achieving secure, efficient, and scalable deployments.

High Gas Fees from Inefficient Storage

Gas fees remain a fundamental challenge for EVM-based apps, impacting both users and developers. Every transaction incurs costs, and for apps processing frequent or complex operations, these fees can quickly add up. If an app relies on a paymaster model, for example, the operator may face substantial expenses. Alternatively, users bear the cost, potentially limiting adoption and engagement.

---

1 "Ethereum Virtual Machine Challenges: Tackling the Hurdles in the EVM Landscape" Jordan Adams, published on Doubloin in November 2023

2 Wolf Game White Paper V2 addressing smart contract issues in November 2021

Optimizing storage and execution techniques is critical to reducing these costs. Efficient onchain storage methods, along with procedural optimizations for nested and cascading operations, has proven to significantly lower gas fees. Without these improvements, apps may end up paying 10x more than necessary in transaction costs, making economic scaling a key consideration for any network.

Challenges in Data Provenance and Ownership Tracking

Managing data ownership and “provenance” – knowing where data originates and ensuring its accuracy – presents its own set of challenges, especially when distinguishing between onchain and offchain data storage. Metadata, which describes assets (e.g., attributes like color or rarity, or traits like age or score), is often stored offchain in centralized databases. While, as mentioned, this can optimize costs, it also introduces trust issues, limits transparency, and creates access barriers. Storing metadata onchain helps eliminate these roadblocks, providing greater transparency, integrity, and seamless data availability.

For example, in a game, if a character and their sword are sold, tracking their ownership history – whether the sword is still tied to the character or owned separately – is crucial for ensuring accurate provenance. Storing this metadata onchain guarantees the integrity and transparency of these relationships, making it accessible across different platforms. Platforms like OpenSea can also leverage this onchain metadata to dynamically display asset details, improving discoverability and enabling greater interoperability. However, this approach typically increases gas costs and requires careful design, making onchain data management a challenging decision in app development.

Lack of Seamless Onchain App Interoperability

Achieving seamless interoperability, where apps can read contracts from other apps and build on top of them, is also a significant challenge in the EVM space. This is largely because most apps store their metadata offchain in order to prioritize scalability.

For instance, ERC-721 tokens are fundamentally not interoperable because their metadata is stored externally, making it difficult for different contracts to access or share data. If a developer wishes to fork an app or extend its functionality, they typically need to rely on external sources to access the contracts and trust that the provided code is accurate and up-to-date. This introduces friction and potential risk, as there is no standardized way to verify or integrate data from one contract to another directly on-chain.

Overall, without standardized onchain metadata schemas, interoperability among EVM-based apps will remain a barrier to ecosystem growth and innovation.

Patchwork's Solution

Patchwork is the ultimate framework for building rich onchain apps. It tackles the key challenges of EVM-based app development, unifying fragmented ideas into a cohesive ecosystem of opportunity. At its core, Patchwork combines a powerful protocol, metadata standards, and specialized tools – all streamlining the necessary elements for developers to build scalable and efficient apps.

How does it work?

Patchwork started off with the idea of "patching" or allowing you to permissionlessly affix, or "soulbind," datasets to any onchain entity or create them as standalone properties. From there, it evolved into an entire developer stack where app data is structured with `Patchwork721` contracts and tokens, which function like database tables: contracts define schemas (columns), and tokens represent rows with unique IDs as primary keys. Seamless protocol orchestration ensures secure, efficient, and queryable data lifecycles across the EVM, streamlining the development of robust apps.

Key benefits

- **Solving EVM app complexities via Automated Smart Contract Generation**
  Patchwork's rule-based Protocol enables automatic contract generation for relational and operational logic, eliminating the need for developers to write extensive boilerplate code and ensuring rapid, secure deployment.

- **Efficient Gas Fee Management**
  Patchwork's byte-packed metadata standard ensures gas-efficient storage and querying, allowing apps to minimize costs while maintaining onchain integrity. While gas remains cheap on some networks today, Patchwork is built with the future in mind, optimizing for a scenario where costs rise, ensuring sustainable, low-cost scaling for apps.

- **Dynamic Ownership and Provenance**
  With support for ownership hierarchies and dynamic metadata, Patchwork enables seamless tracking and management of complex asset relationships – whether it's linking a game character to its equipment or validating ownership histories.

- **Schema-Driven Interoperability**
  Standardized, discoverable schemas allow apps to seamlessly share and integrate data. This eliminates the silos that limit ecosystem collaboration and simplifies the creation of collaborative onchain systems and overall data availability.

- **Permissionless Extensibility**
  Developers and third parties can build on and extend existing apps freely, fostering innovation, decentralized collaboration, and ecosystem growth without compromising integrity or control.

- **Ensuring Security**
  Patchwork contracts optimize for maximal asset security, e.g., with locking and freezing mechanisms. The Patchwork Protocol and base contracts that the PDK use have gone through multiple rounds of auditing by Macro[3][4], and code libraries that are included by our code are based on audited OpenZeppelin contracts.

---

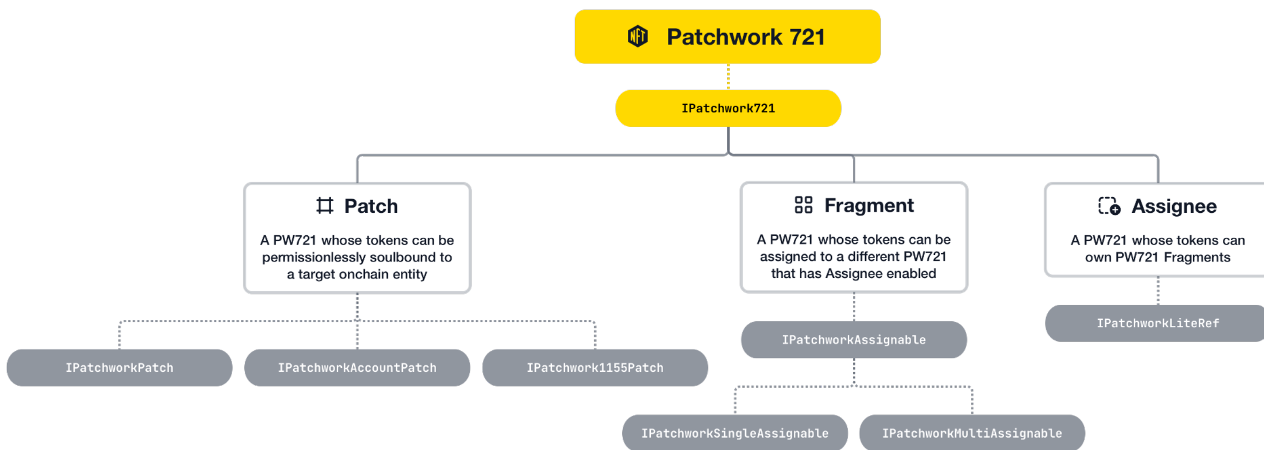
3 Macro Patchwork security audit in 2023

4 Macro Patchwork security audit in 2024

Core Concepts

The Patchwork721 Standard

The Patchwork721 Standard is a core component of Patchwork's framework, leveraging the `IPatchworkMetadata` interface to enable dynamic interactions between assets while optimizing efficiency, interoperability, and reducing gas fees. In short, `Patchwork721` extends the classic `ERC-721` beyond a simple asset standard, unlocking new possibilities for app use cases & data tokenization.



THE PATCHWORK721 STANDARD

Patches and Fragments

Patches, a macro extension contract of `Patchwork721`, are like personal annotations in a decentralized knowledge graph. While they don't alter the original onchain entity - be it a contract address, wallet, or token - they create a parallel layer of context and functionality that can easily be discovered via the Patchwork explorer.

Much like how indexing protocols surface hidden metadata across Web3, Patches enable permissionless collaboration without modifying the original source.

Fragments, also a macro extension contract, serve as modular building blocks within the Patchwork ecosystem. These discrete data units can be assigned to or unassigned from a parent `Patchwork721` and traded independently.

Ownership models of Fragments can be managed in different ways: →

- **Proxied (or Hierarchical) Ownership:** Like having a keyholder for a group of items, when the keyholder (the parent) changes, the ownership of everything tied to it changes too, without needing to move each item individually. As an optimization, `ownerOf()` is proxied so no transfers actually happen. Patchwork instead emits cheap transfer logs onchain so indexers and explorers are made aware of the ownership change.
- **Weak References:** This allows Fragments to be assigned to an entity without transferring ownership, like selling a piece of a puzzle, but that piece will always stay part of the puzzle.
- **Transferability:** This controls whether Fragments can be locked, moved, or made permanent, ensuring assets can be transferred or remain fixed, such as for permanent onchain audit reports.

Assignees and Scopes

Assignees (`LiteRefs`) and Scopes work together to enable dynamic asset management and application organization. Assignees are specialized contracts designed to hold and manage Fragments efficiently, using a storage system called LiteRefs. Normally, to uniquely address an `ERC-721` token, it requires 512 bits of data: 256 bits for the address and 256 bits for the token ID.

However, Patchwork takes a practical approach by compressing this reference down to just 64 bits, resulting in an 8x savings on storage. Think of an Assignee as a backpack that organizes and carries gear, with defined slots that hold specific Fragments. When transferred, an Assignee moves with its entire tree of assigned Fragments, ensuring seamless asset portability.

Scopes, on the other hand, act as reserved namespaces for developers, providing a foundation to organize and govern their apps. Like owning a plot of land, a Scope defines where app-specific rules, permissions, and connections to protocol utilities reside. Scopes also enable developers to accrue fees from their contracts, offering a streamlined way to manage dApps within the ecosystem.

Locks and Freezes

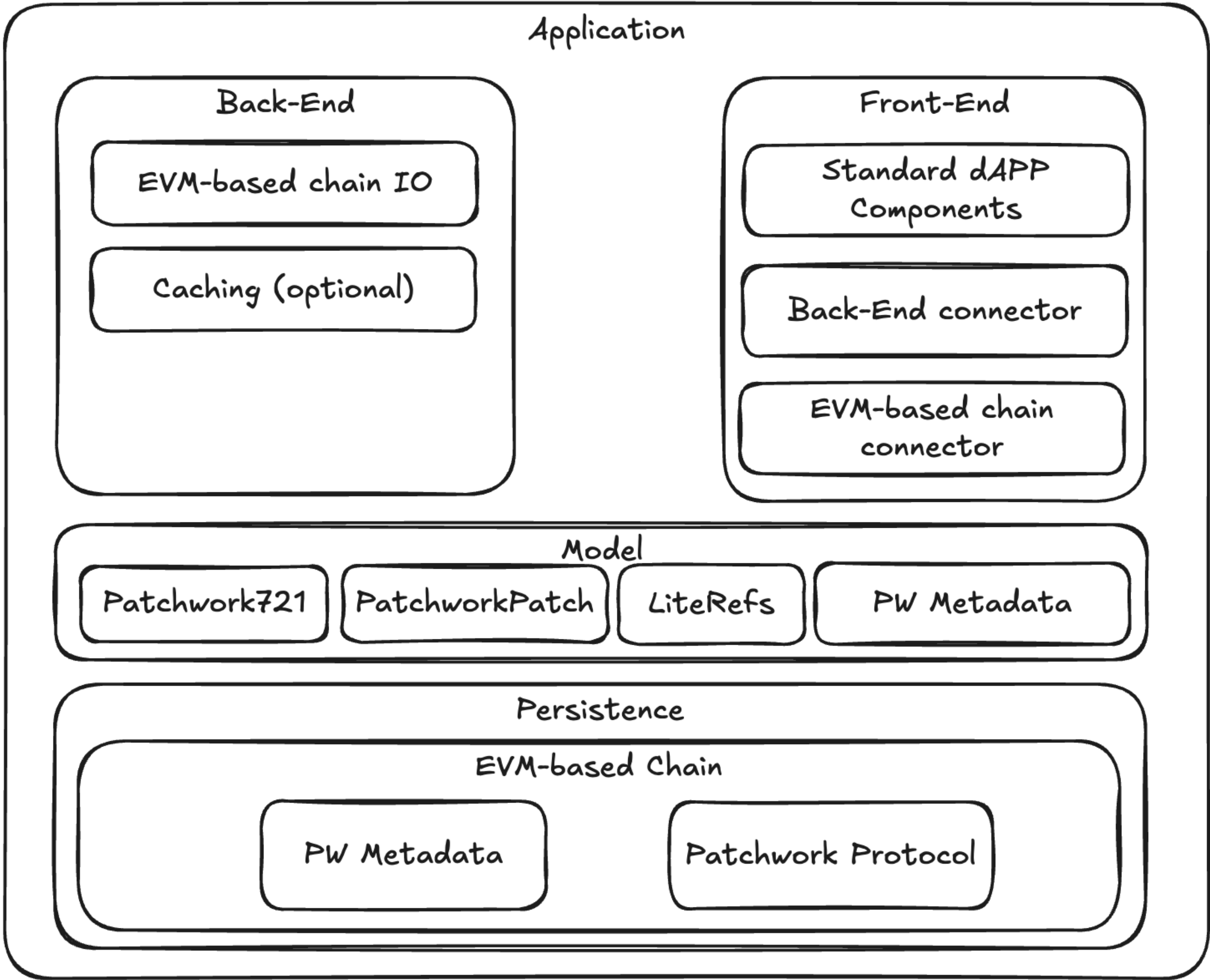
Patchwork's robust security framework incorporates two essential mechanisms for protecting asset integrity and preventing unauthorized modifications: locking and freezing. The locking mechanism, which implements `IERC-5192`, provides transfer restrictions by preventing token movement until explicitly unlocked, with all state changes broadcast through standardized Locked and Unlocked events.

Complementing this, the freezing feature allows for programmatic freezing of assets (like owned NFTs) using a nonce that changes every time a thaw occurs, allowing a check for a nonce match on sale. If the nonce does match, the token you're purchasing could have been tampered with.

These mechanisms are particularly crucial in the context of composable assets, where the integrity of inter-token relationships and associated data must be guaranteed during transfers to prevent potential "rug pulls" or unauthorized modifications to the asset's composition.

The Patchwork Stack: Simplifying App Development

Building on these core technical concepts, the Patchwork Stack is a fully integrated framework, where each component works seamlessly together to simplify efficient onchain app development.



THE PATCHWORK STACK

Base Contracts

At the very start of Patchwork is its base contracts, which define relationships, permissions, and transfer mechanics, ensuring modular and composable interactions across apps. As mentioned in Core Concepts, all our contracts build upon and extend the `Patchwork721` standard, which serves as the fundamental primitive throughout the ecosystem.

- **`Patchwork721`**
  An extension of ERC-721 and the core contract that all other Patchwork contracts extend from.
- **`Patchwork1155Patch`**
  Targets specific accounts or an entire token ID.
- **`PatchworkPatch`**
  A Patchwork721 contract that links to another 721's address and token ID.
- **`PatchworkAccountPatch`**
  Attaches directly to a user's wallet address, allowing account-level extensions.
- **`PatchworkFragmentMulti`**
  Enables a fragment to be assigned to multiple holders simultaneously.
- **`PatchworkFragmentSingle`**
  Enables a fragment to be assigned to a single holder.

Each contract type adheres to a specific Patchwork interface and can be generated through the Patchwork Developer Kit (PDK) with corresponding configurations.

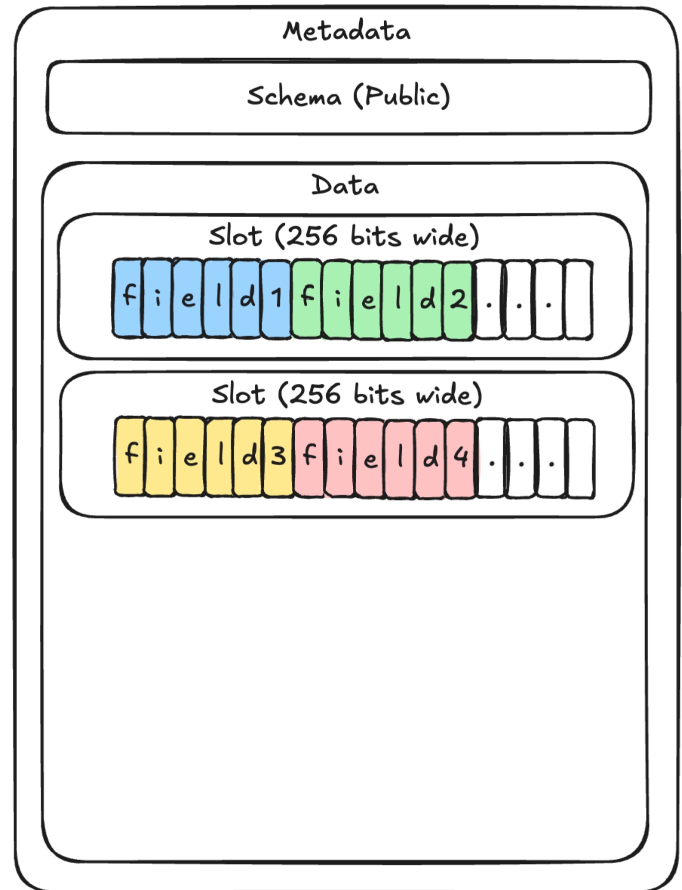
Data Modeling: Metadata Standard

Patchwork's Metadata Standard is the backbone of enabling interoperability and efficient data management across apps, defining a schema-based system where each piece of data is organized and stored in a format that ensures consistency. It can also be used by apps not leveraging Patchwork to allow their data model to be discoverable onchain via `schema()`.

On top of this, Patchwork automatically offers maximal efficient layouts, with the metadata being packed into uint256 values. This allows developers to group frequently accessed fields into a single `SSTORE` and reads into a single `SLOAD`, which the compiler

would not do for you when using a struct directly. For example, instead of using separate slots for metadata fields like ownership, attributes, and relationships, these can be combined into a single packed slot, saving gas per `SSTORE` and per `SLOAD`.

Orchestrated by the Patchwork Protocol and implemented by the PDK, this system ensures Patchwork-based apps are easily scalable and adaptable according to the schema they put in place.

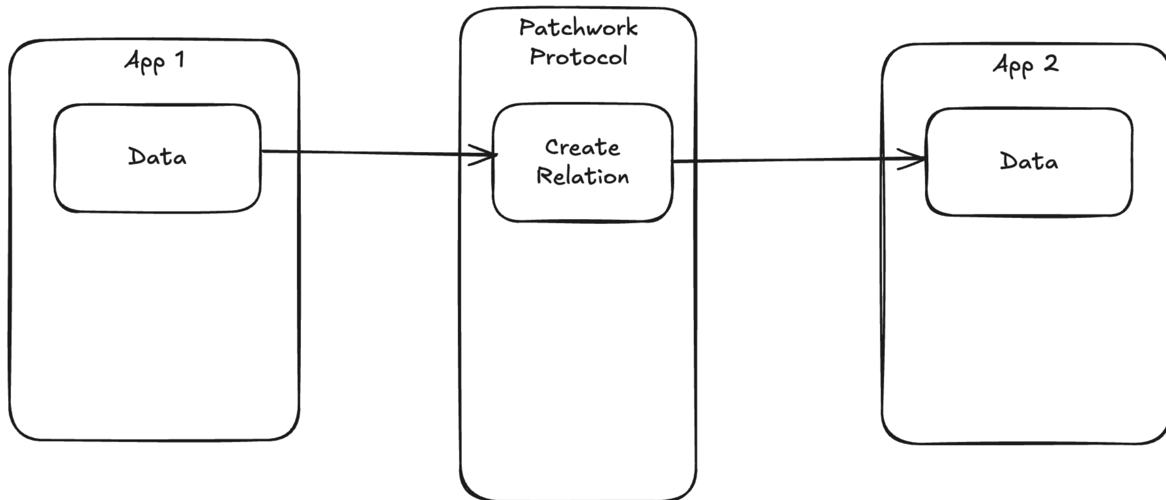


PATCHWORK METADATA SCHEMA

Defining Relationships: Patchwork Protocol

The Patchwork Protocol acts as the orchestrator of onchain metadata relationships, implementing the core technical contracts that form the backbone of the ecosystem. By doing so, it ensures data integrity and rule adherence across connected entities. Think of it as a referee in a game, making sure all teams follow the rules and that data interactions are fair and secure.

By managing these relationships, it enables seamless, trust-based connections between contracts, accounts, and tokens. With the added benefits of automatic ownership model updates, fee management, and accounting, the Protocol simplifies complex app development while protecting against bad actors and ensuring reliability.



PATCHWORK PROTOCOL ARCHITECTURE

Developer Tooling

Patchwork Development Kit (PDK)

The Patchwork Development Kit (PDK)[5] empowers developers to create and deploy sophisticated onchain apps with minimal effort. Just as a well-stocked toolkit provides all the essential tools for assembling something, the PDK bridges the gap between concept and execution. Developers simply define their app's requirements in a configuration file, and the PDK generates contracts tailored to those needs.

5 PDK: https://docs.patchwork.dev/pdk/overview

The kit supports various configurations, including complex metadata fields, relational data structures, and application-specific logic. Beyond generating contracts, the PDK can also produce backend setups for tools like Ponder, enabling developers to go from configuration to a running app seamlessly.

PDK Workflow

1. **Define Configuration**
   Developers outline their contract requirements in a typescript file, specifying features, metadata fields, and relationships.
2. **Generate Contracts**
   The PDK converts the configuration into solidity contracts, reflecting the specified parameters.
3. **Customize Logic**
   The generated contract can be further refined with application-specific logic for unique use cases.
4. **Deploy to Blockchain**
   The finalized contract can be deployed to any EVM-compatible blockchain where the Patchwork Protocol operates.
5. **Interact via PDK**
   Developers use PDK-generated functions to interact with the contract, integrating seamlessly with the broader Patchwork ecosystem and easily discoverable via the Explorer.

```
import { ContractConfig, Feature, FunctionConfig, ProjectConfig } from "@patchworkdev/common/types";

const myConfig: ProjectConfig = {
    name: "Contract Config Project",
    scopes: [
        {
            name: "test",
            owner: "0x222222cf1046e68e36E1aA2E0E07105eDDD1f08E",
            whitelist: true,
            userAssign: false,
            userPatch: false,
            bankers: ["0x000000254729296a45a3885639AC7E10F9d54979", "Contract1"],
            operators: ["0x000000111129296a45a3885639AC7E10F9d54979", "Contract1"],
        }
    ],
    contracts: {
        "Contract1": {
            scopeName: "test",
            name: "AccountPatch",
            symbol: "AP",
            baseURI: "https://mything/my/",
            schemaURI: "https://mything/my-metadata.json",
            imageURI: "https://mything/my/{tokenID}.png",
            fields: [
                {
                    id: 1,
                    key: "name",
                    type: "char32",
                    description: "Name",
                    functionConfig: FunctionConfig.ALL,
                },
                {
                    id: 2,
                    key: "patches",
                    type: "literef",
                    description: "Contract2",
                    arrayLength: 4,
                }
            ],
            features: [Feature.ACCOUNTPATCH],
            fragments: ["Contract2"]
        },
        "Contract2": {
            scopeName: "test",
            name: "SecondContract",
            symbol: "SC",
            baseURI: "https://mysecondthing/my/",
            schemaURI: "https://mysecondthing/my-metadata.json",
            imageURI: "https://mysecondthing/my/{tokenID}.png",
            fields: [
                {
                    id: 1,
                    key: "description",
                    type: "char32",
                    description: "Description",
                    functionConfig: FunctionConfig.ALL,
                }
            ],
            features: [Feature.PATCH, Feature.FRAGMENTSINGLE],
            fragments: []
        }
    },
    plugins: [{ name: "ponder" },{ name: "react" }],
};

export default myConfig;
```

EXAMPLE PATCHWORK CONFIGURATION

Create-Patchwork

To further streamline and speed up the setup process for Patchwork-based projects, the Create-Patchwork[6] CLI automates key tasks, serving as a primary entry point for developers. Leveraging the PDK, the tool generates all necessary code, orchestrating backend services and frontend integrations from a single configuration. This includes smart contract generation, backend setup, and the configuration of services such as Docker containers and other essential components.

The CLI tool also includes built-in support for Ponder (event indexing), a preconfigured web3-ready React frontend (with TailwindCSS and RainbowKit), and PostgreSQL for data persistence, delivering a fully-configured development environment. By automating the generation of smart contracts, deploy scripts, APIs, and React hooks, it accelerates the development process and eliminates the need for manual configuration.

Patchwork Wizard

Developers can also experiment with Patchwork and its PDK directly by using the Patchwork Wizard[7]. With text-to-config implemented, simply provide a text description of your app's schema, and the Wizard will generate the smart contracts in real time, automatically integrating them into Create-Patchwork to kickstart your app development.

Patchwork Explorer

Finally, there's the Patchwork Explorer[8]. This is the gateway to the Patchwork ecosystem, providing a clear view of the different apps, contracts, and interactions within the network, making it easy to discover, analyze, and track onchain activity.

At the time of publishing, Explorer indexes 12 apps and 90 contracts across Base mainnet and testnet, displaying key stats such as mints, unique addresses, assignments, data rows, and interactions. As the Patchwork ecosystem grows, Explorer will continue to evolve as a hub for exploring composable onchain apps and their respective data.

6 create-patchwork: https://docs.patchwork.dev/pdk/getting-started

7 Patchwork Wizard: https://wizard.patchwork.dev

8 Patchwork Explorer: https://explorer.patchwork.dev

Example Use Cases

Composable Game Items

When you're playing a game and your character owns a sword, what happens to the sword if you sell the character? Patchwork unlocks new possibilities for Web3 game development by enabling modular and interconnected game assets.

Imagine creating a game where characters own armor, trophies, or pets that can be traded, upgraded, or soulbound. Patchwork’s hierarchical ownership model ensures that when a player sells a character, linked items like swords automatically transfer ownership, while soulbound items, such as pets, remain attached. Developers can easily define these relationships using Patchwork’s schema-based tools.

This approach reduces development complexity by providing a unified framework for managing modular assets. It also enhances player experience by streamlining asset interoperability and enabling seamless trading across games. Compared to traditional methods, Patchwork accelerates development and fosters innovation, allowing developers to focus on creating engaging gameplay.

Onchain Transactions & Rewards

Managing complex, interdependent data onchain becomes straightforward with Patchwork. For example, in e-commerce, Patchwork can attach receipts as onchain metadata to user accounts, ensuring tamper-proof, transparent, and updatable transaction histories. This eliminates concerns about data integrity and simplifies backend workflows.

Similarly, in event ticketing, Patchwork resolves challenges like verifying ticket authenticity and enabling resales by linking tickets to their

corresponding events. For schools, Patchwork's dynamic data management supports use cases like real-time updates to student attendance and grades across multiple classes.

By offering a composable design, Patchwork turns intricate data relationships into scalable, efficient solutions, enabling developers to build reliable apps with ease while ensuring transparency and trust for end-users.

Verified Trust Systems

Speaking of trust, it's a cornerstone of Web3, yet building reliable trust mechanisms often involves fragmented or inefficient solutions. Patchwork addresses this challenge by enabling dynamic, onchain trust systems that are transparent, interoperable, and immutable.

For example, in smart contract auditing, developers can use Patchwork to attach signed audit records directly to contracts as metadata. These records provide a tamper-proof verification trail, enabling programmatic trust mechanisms such as restricting unverified contracts from critical operations. Unlike offchain alternatives, Patchwork's approach ensures universal accessibility and immutability.

Patchwork also empowers ID verification systems by allowing users to attach KYC attestations to their accounts. With built-in features for updates and revocation, developers can maintain compliance while preserving data sovereignty. These capabilities elevate trust and transparency across diverse Web3 apps.

Soulbound Tokens

Patchwork's soulbound tokens redefine how developers create permanent, non-transferable connections between onchain assets, accounts, and metadata. For instance, proof of participation badges issued to DAO voters or hackathon attendees can serve as immutable records of involvement. Contributor rewards tied directly to Ethereum accounts ensure that recognition reflects genuine contributions rather than being sold on secondary markets.

Unlike traditional NFTs, Patchwork enforces immutability at the contract level, ensuring soulbound tokens cannot be transferred or removed once assigned. This feature simplifies the creation of apps for reputation, certification, and recognition, offering a secure and transparent way to establish trust and authenticity. Developers can leverage this functionality to build innovative systems for verifying achievements, issuing credentials, and fostering long-term user engagement.

Roadmap

With Patchwork's foundation firmly established – featuring a robust Protocol, PDK, the interactive Wizard, and Explorer – the platform is prepared to take EVM app development to the next level.

The next step is realizing Patchwork's ultimate vision: transitioning from low-code to no-code. This means enabling anyone to simply describe their app, and Patchwork laying the foundation to seamlessly generate fully interoperable smart contracts and backend infrastructure entirely onchain – ready to deploy. Like Wordpress for apps.

To achieve this, upcoming advancements like analytics, cross-app discovery, and distribution tools will expand capabilities. The community has already been a key driver in Patchwork's development, and future plans include fostering more interaction and building on Patchwork community ideas. Looking further ahead, Patchwork looks forward to driving a vibrant ecosystem with token-based rewards, fee-sharing models, and a dynamic portal for app discovery and inter-app engagement.

Conclusion

Get Started With Patchwork

Patchwork revolutionizes the way developers build EVM-based apps, offering a powerful, flexible solution that streamlines complex data management, reduces operational costs, and unlocks the full potential of all kinds of Web3 apps to build and interact with each other.

Whether you're building your first EVM-based app or advancing an existing project, Patchwork is your best toolkit. Start building with Patchwork today and easily turn your vision into reality: https://docs.patchwork.dev/protocol/getting-started/101

Who's Behind Patchwork

Patchwork was created by the team behind Paradex, one of the first decentralized exchanges (DEXs) in the crypto space, which was acquired by Coinbase in 2018. Following the acquisition, we worked on integrating decentralized features into Coinbase, blending innovation with scalability. Our team – including Kevin Day, Rob Green, Brock Petrie, and Don MacLellan – brings a wealth of expertise across backend and frontend engineering, game design, and the development of decentralized financial models.

Patchwork is funded by leading investors, including Base Ecosystem Fund and IDEO Ventures, a globally respected design and innovation firm. Patchwork also is being advised by Ron Bernstein who previously founded AugmentPartners, a software development company dedicated to applying decentralized protocols to real-world financial systems. Ron also led Periscope Trading, a specialized division of 0x Lab. This backing highlights both the technical credibility and creative potential of the Patchwork vision.

Disclaimer

While the Patchwork Protocol and base contract code were audited and generated code is generally reliable, publishers are ultimately responsible for conducting independent audits to address any security concerns. As Patchwork continues to evolve, please be aware that the generated code is regularly updated.