Hyperweb: A Universal TypeScript & JavaScript Virtual Machine for the Internet of Blockchains

Dan Lynch, Simon Warta, Anmol Yadav, Jeet Raut, Phat Gia Ha, Luca Chang, Zhi Zhen, Lucas Chiang, Eason Huang, Gefei Hou, Jovonni Pharr

info@hyperweb.io

v0.0.1 - February 5, 2025

Abstract

The fragmentation of blockchain ecosystems–driven by esoteric languages, distinct toolchains, and the isolation of Web3 from the broader software stack–creates a high barrier to entry, and limits developer participation. This challenge also confines use cases to typical blockchain applications observed in the market to date, and makes applications inherently harder to scale. Hyperweb unifies this experience by providing a complete, end-to-end TypeScript ecosystem for building decentralized applications, combining infrastructure, client tools, and smart contract frameworks. This paper details a novel approach towards enabling powerful blockchain application development using the World's most popular programming language [1], expanding the developer base and accelerating adoption. We present the Hyperweb Virtual Machine (HVM), a novel virtual machine architecture for executing smart contracts written in TypeScript and JavaScript. The protocol serves as an Interchain JavaScript Hub to provide developers methods to control actions on other chains via the HVM. The stack encompasses several components working together to serve as a multifaceted solution to a cohesive and universal web development experience.

Contents

1 Introduction

The blockchain ecosystem has evolved significantly, yet development remains complex and error-prone. Hyperweb addresses this challenge by providing a unified virtual machine that enables developers to write smart contracts in TypeScript and JavaScript, languages familiar to millions of developers worldwide. This paper presents the architecture and implementation details of the Hyperweb Virtual Machine (HVM), focusing on its key innovations in state management, cross-chain execution, and developer experience.

1.1 Ecosystem

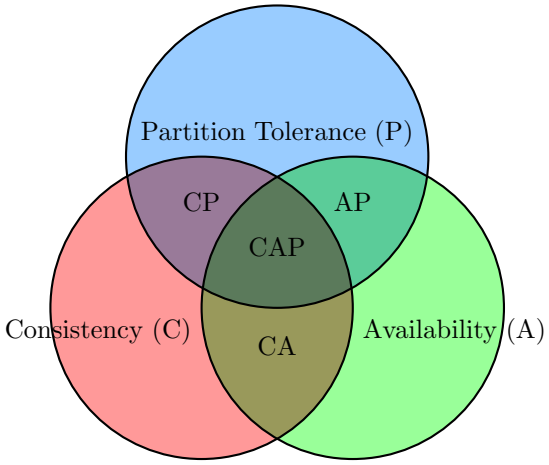
Hyperweb's ecosystem has made a significant impact on the Web2 and Web3 developer landscape, with over **40 million total downloads** across its tools. Hyperweb's Web2 tools have been downloaded **10.7 million times**, while Web3 tools lead with **27.6 million downloads**, demonstrating strong adoption across blockchain ecosystems. Utility libraries have also seen substantial traction, with **1.89 million downloads**, streamlining developer workflows. Key tools such as Cosmos Kit (**1.58M+ downloads**), Telescope (**5.29M+ downloads**), and Chain Registry (**6.09M+ downloads**) have become essential for multi-chain application development. The Chain Registry further highlights Hyperweb's reach, supporting well over **200 chains** [2]. Notably, Supabase, one of the platforms leveraging Hyperweb's database tooling, now manages nearly **2 million databases**, adding approximately **2,500 databases** per day [3]. These figures underscore Hyperweb's role in bridging Web2 and Web3 development, lowering the barrier to entry, and enabling developers to build scalable, interoperable applications with ease [4].

1.2 CAP Theroem

The **CAP theorem**, introduced by Eric Brewer in his keynote at the 2000 PODC conference [5], states that a distributed system can provide at most two of the following three guarantees simultaneously:

1. **Consistency (C):** Every read receives the most recent write or an error.
2. **Availability (A):** Every request receives a response, even if some nodes fail.
3. **Partition Tolerance (P):** The system continues to operate despite network partitions.

In practical terms, no distributed system can achieve all three properties fully; designers must choose which two are most critical to their application, acknowledging trade-offs.



1.3 Shortage of Web3 Developers

One major hurdle facing blockchain ecosystems is the relative scarcity of experienced Web3 developers compared to the vast pool of traditional Web2 engineers. The specialized skill set required—ranging from smart contract programming and security audits to cryptographic key management—creates a high barrier to entry. Moreover, the fragmented toolchains and distinct language dialects across different chains (e.g., Solidity, Rust, Move) deter many Web2 developers. Hyperweb aims to alleviate this bottleneck by leveraging the ubiquitous familiarity of JavaScript and TypeScript, thereby reducing onboarding friction. By unifying cross-chain execution under a single, accessible VM model, Hyperweb broadens the potential developer base and accelerates adoption.

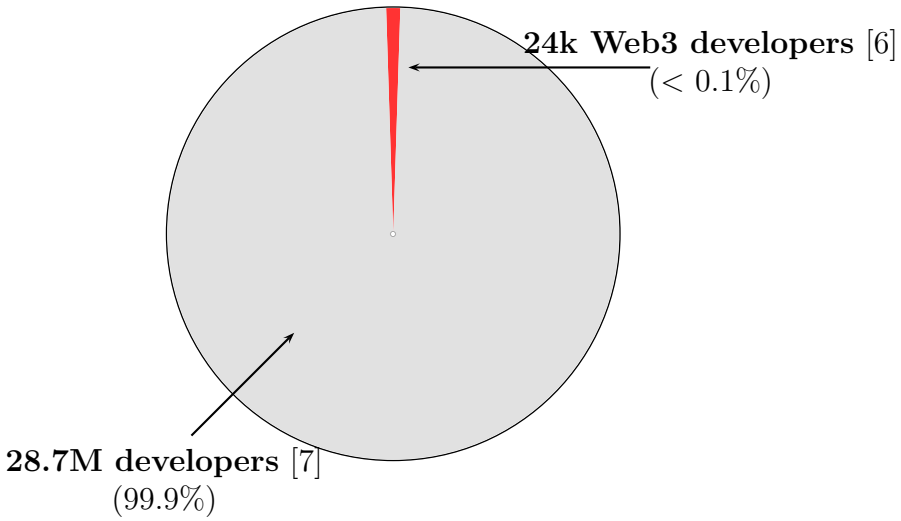


Figure 1: Estimated ratio of active Web3 developers to all developers, illustrating the scarcity of blockchain-savvy engineers.

1.3.1 Esoteric Programming Languages and Frameworks in Blockchain Ecosystems

Challenge: There are 28.7 million developers in the world [7], yet blockchain ecosystems rely heavily on esoteric programming languages and frameworks that require deep, chain-specific expertise. This limits participation and discourages developers who are unfamiliar with these niche tools. As a result, there are only 24,000 active Web3 developers globally [6], representing less than 0.1% of the total developer population.

Solution: Hyperweb bridges this gap by offering a JavaScript virtual machine and TypeScript smart contract environment. This solution empowers the global community of 20 million JavaScript developers [1, 7] to engage with blockchain technology, removing the steep learning curve posed by esoteric languages.

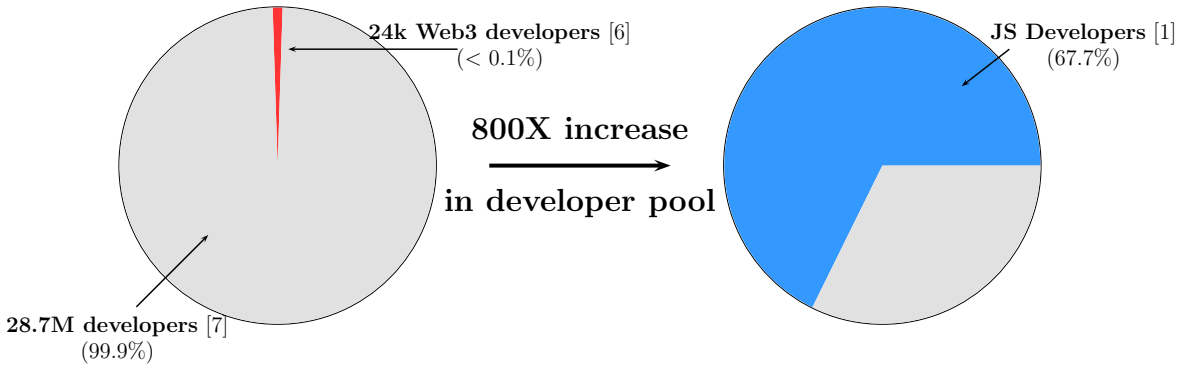


Figure 2: Estimated ratio of active Web3 developers to JavaScript (Web2) developers, illustrating the opportunity to increase in developer population.

Define the sets:

$$S_{\text{Web3}} : \text{Set of active Web3 developers,}$$
$$S_{\text{JS}} : \text{Set of active JavaScript developers,}$$
$$f : S_{\text{JS}} \to S_{\text{Web3}} : \text{Mapping that enables JavaScript developers to become Web3 developers.}$$

For any developer $d \in S_{\text{JS}}$, let $f(d) = d'$, where:

$$d : \text{A developer skilled in JavaScript,}$$
$$d' : \text{The same developer, after acquiring Web3 development skills through } f.$$

Introduce a parameter Δ, where:

$$\Delta \in [0, 1], \quad \Delta = 1 \implies \text{Perfect alignment with JavaScript syntax (no additional learning curve),}$$

and

$$\Delta = 0 \implies \text{Complete deviation from JavaScript syntax (entirely new concepts to learn).}$$

The transformation induced by f depends on Δ, where:

$$S'_{\text{Web3}} = f(S_{\text{JS}}, \Delta),$$

and each $d' \in S'_{\text{Web3}}$ corresponds to a $d \in S_{\text{JS}}$ transformed with a cognitive effort proportional to $1 - \Delta$.

Initially, we observe:

$$S_{\text{Web3}} \subset S_{\text{JS}}, \quad \text{and} \quad |S_{\text{Web3}}| \ll |S_{\text{JS}}|.$$

After applying f, the expanded set satisfies:

$$S_{\text{Web3}} \subseteq S'_{\text{Web3}} \quad \text{and} \quad |S'_{\text{Web3}}| \approx |S_{\text{JS}}| \cdot \Delta.$$

Thus, the effectiveness of the transformation f and the size of S'_{Web3} are directly influenced by Δ.

1.4 Developer Experience Challenges

The developer experience in Web3 presents several significant challenges that hinder rapid adoption and efficient application development. Below, we outline these challenges and how our approach seeks to resolve them.

1.4.1 Fragmented Ecosystem

Challenge: The Web3 ecosystem consists of a variety of blockchains, each with its unique architecture, tooling, and programming languages. Developers face steep learning curves when switching between chains or working on multichain applications.

Solution: By introducing Interchain JavaScript, we unify the development experience across heterogeneous chains. Developers can write applications in a familiar language while leveraging shared tooling for contracts and frontends.

1.4.2 Limited Interoperability

Challenge: Ensuring seamless communication between different blockchains remains a significant hurdle, requiring developers to handle complex cross-chain messaging and state synchronization manually.

Solution: Interchain JavaScript and the HVM will provide a unified execution environment that abstracts cross-chain interactions, enabling developers to focus on application logic rather than infrastructure concerns.

1.4.3 Poor Tooling and Documentation

Challenge: Developers often encounter poorly documented tools and fragmented resources, making it difficult to onboard new teams or scale development efforts.

Solution: Our comprehensive suite of tools, including InterchainJS and Hyperweb, comes with detailed documentation and tutorials, ensuring a smooth onboarding experience for developers of all skill levels.

1.4.4 Security Risks

Challenge: Building secure applications requires developers to be aware of numerous pitfalls, including reentrancy attacks, mismanaged keys, and vulnerabilities in third-party dependencies.

Solution: Our framework incorporates built-in security primitives, such as security tooling and safe contract templates, reducing the likelihood of common vulnerabilities.

1.4.5 Cross-Chain Application Complexity

Challenge: Developers must write, deploy, and maintain contracts separately for each blockchain, adding complexity in heterogeneous ecosystems with distinct languages and tools.

Solution: A write-once, run-everywhere approach enables developers to write contracts in Interchain JavaScript and execute them across multiple chains using HVM, significantly reducing complexity and maintenance overhead.

1.4.6 Web2 and Web3 are Isolated

Challenge: To have a truly scalable, accessible, and decentralized system, different parts of the infrastructure stack must leverage different technologies. For example, in Web3, indexers are typically added as an after thought to address the cap theorem 1.2.

Solution: Treating technologies such as indexers as first class primitives abstracts away complexity, enabling teams to create event-driven applications that handle everything from off-chain computation to cross-chain transactions with the same intuitive developer experience. The result is a unified full-stack ecosystem that combines Web2's proven performance with Web3's powerful composability, making decentralized development feel natural and infinitely scalable.

1.4.7 Motivation

By addressing these challenges, we aim to create a developer-centric ecosystem that fosters innovation and simplifies the development of scalable, secure, and interoperable Web3 applications.

1.5 Web3 in the Context of the CAP Theorem

Using the **CAP theorem**, we can describe Web3 and its relationship with Web2 as follows:

Web3 as C and P

Web3 prioritizes:

- **Consistency (C):** Ensuring verified and tamper-proof data through decentralized consensus mechanisms.
- **Partition Tolerance (P):** Maintaining decentralized data that continues to operate under network partitions or disruptions.

However, by focusing on C and P, Web3 systems often sacrifice **Availability (A)**, as achieving consistency in a decentralized network introduces latency (e.g., block finality and consensus delays).

Web2 as C and A

Web2 systems prioritize:

- **Consistency (C):** Delivering reliable, consistent responses from centralized databases.
- **Availability (A):** Ensuring responsiveness and low-latency experiences.

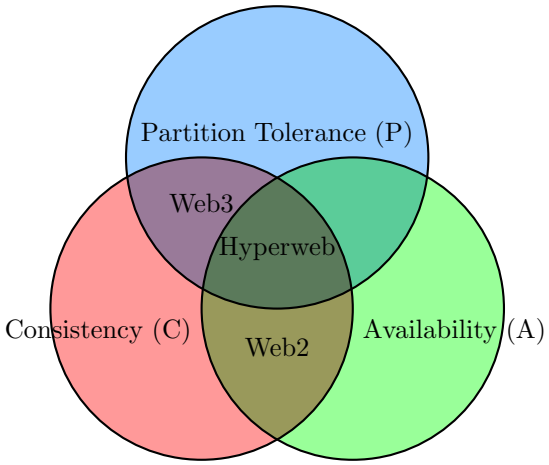
Web2 systems deprioritize **Partition Tolerance (P)**, as centralization provides resilience through redundancy rather than decentralization.

Web2 Supporting Web3 Scalability

Web2 technologies, such as indexers and caching solutions, act as intermediaries that inject **Availability (A)** into Web3 workflows. These tools:

- Organize and cache data from decentralized networks to provide fast and scalable access for read operations.
- Enable scalable and efficient access to Web3 data for reads, while preserving C (Consistency) and P (Partition Tolerance) during write operations.

By properly delegating read optimization to Web2 technologies and reserving the rigor of C and P for writes, Web3 achieves scalability without compromising its core principles.



2 Interchain JavaScript

Interchain JavaScript represents a paradigm shift in the development of Web3, unifying heterogeneous blockchain ecosystems under a single, universal programming model. This vision draws from the analogy of JavaScript in the conventional Internet – a language capable of running across all websites, providing universal utility.

In the context of Web3, where blockchains are often referred to as the "Internet of Blockchains", Interchain JavaScript enables seamless interaction across all chains, breaking down interoperability

barriers and facilitating frictionless communication between diverse blockchain ecosystems. By leveraging Interchain JavaScript, developers can focus on crafting innovative solutions rather than grappling with the intricacies of disparate blockchain protocols.

2.1 Core Components of the Interchain JavaScript Stack

Hyperweb Virtual Machine (HVM): At the core of Interchain JavaScript is the HVM, a versatile virtual machine designed to execute JavaScript. Formally, this execution model can be expressed as:

$$\mathcal{E}_{\text{chain}}(\mathcal{P}_{\text{JS}}) \rightarrow \mathcal{R}_{\text{chain}}$$

Where:

- $\mathcal{E}_{\text{chain}}$: Execution environment of a specific blockchain.
- $\mathcal{P}_{\text{JS}}$: JavaScript program to be executed.
- $\mathcal{R}_{\text{chain}}$: Resulting state/output from the execution.

Signing Client (InterchainJS): InterchainJS serves as the core library for signing and interacting with multiple blockchains using an adapter pattern. It simplifies key management and supports universal signing mechanisms, including EIP712 and Cosmos SDK transactions.

SDK Clients (Telescope): Telescope is the frontend companion for InterchainJS. It is a TypeScript transpiler for Cosmos SDK Protobuffers, allowing developers to generate blockchain-specific SDKs for seamless interaction. It enables automatic TypeScript bindings, reducing the complexity of working with blockchain protocols, UI frameworks, and state management.

Wallet Adapters (Interchain Kit): Interchain Kit provides a universal wallet adapter supporting multiple blockchain wallets, such as Keplr, Ledger, and MetaMask. It simplifies wallet connections across multiple networks, including both Ethereum and Cosmos ecosystems.

Unified Frontend Framework (Interchain UI): A cross-framework library of UI components tailored for interchain apps with first-class support for React and Vue. Underlying compiler allows to compile to other UI frameworks.

Starter Kits (Create Interchain App): Developers can set up a modern Interchain app in seconds with `create-interchain-app` by running a single command. Projects come preloaded with InterchainJS, Interchain Kit, and Interchain UI.

Chain Information (Chain Registry): The Chain Registry is a unified repository that provides standardized blockchain metadata, including token symbols, logos, and IBC denominations. It streamlines chain discovery and integration into decentralized applications.

Testing Frameworks (Starship): Starship is a unified development and testing environment that enables developers to simulate blockchain ecosystems. It facilitates end-to-end testing of interchain applications, improving reliability and accelerating development cycles.

Composability Across the Stack: The architecture enables seamless integration of smart contracts, user interfaces, and data layers, abstracting away blockchain-specific complexities. This modular approach is represented as:

$$\mathcal{A} = \{(\mathcal{C}_1, \mathcal{U}_1), (\mathcal{C}_2, \mathcal{U}_2), \ldots, (\mathcal{C}_n, \mathcal{U}_n)\}$$

Where $\mathcal{A}$ is the application composed of n contract-interface pairs.

2.1.1 Key Advantages of Interchain JavaScript

Universal Compatibility: By enabling JavaScript to execute universally, developers are no longer constrained to ecosystem-specific languages. This compatibility ensures:

- Consistency in application logic.
- Reduced learning curves for developers.

Developer-Centric Approach: Interchain JavaScript allows developers to write in a familiar language across the entire Web3 stack, from smart contracts to frontends. The unification of tooling eliminates fragmentation and accelerates innovation.

Scalability and Extensibility: The modular architecture supports seamless scaling by adding new components or chains, preserving the integrity of the global application logic:

$$\mathcal{A} \to \mathcal{A}' \quad \text{where} \quad \mathcal{A}' = \mathcal{A} \cup \{(\mathcal{C}_{n+1}, \mathcal{U}_{n+1})\}$$

Future-Proofing Web3: Interchain JavaScript positions Web3 as a true "Internet of Blockchains," embodying the principles of interoperability, universality, and inclusivity that made the conventional internet successful.

2.1.2 Formalizing the Goal

The vision of Interchain JavaScript can be summarized as:

$$\mathcal{IJS} = (\mathcal{H}_{\text{VM}}, \mathcal{I}_{\text{UI}}, \mathcal{T})$$

Where $\mathcal{IJS}$ combines:

- $\mathcal{H}_{\text{VM}}$: JavaScript virtual machine for execution.
- $\mathcal{I}_{\text{UI}}$: Interchain UI framework for frontend development.
- $\mathcal{T}$: Suite of tools supporting end-to-end application development.

3 System Architecture

3.1 Overview

Hyperweb's architecture consists of four primary components:

- Contract Execution Environment
- State Management System
- Event Processing System
- Cross-Chain Communication Layer

The system is designed to provide deterministic execution across different blockchain environments while maintaining the flexibility and ease of use that TypeScript and JavaScript developers expect.

4 State Management System

4.1 State Storage

The state management system utilizes an Immutable AVL (IAVL) tree structure for efficient state storage and verification. This implementation provides:

- O(log n) access and modification complexity
- Cryptographic verification of state transitions
- Efficient proof generation for cross-chain communication

5 Event System

5.1 Event Architecture

The event system implements a type-safe approach to event emission and handling:

```
type Event = {
    kind: string;
    data: any;
    timestamp: number;
    blockHeight: number;
}
```

Listing 1: Event

5.2 Event Processing

Events are processed through a pipeline that ensures:

- Type safety at compile time
- Efficient serialization for cross-chain communication
- Automatic event attribution in the SDK context

6 Contract Lifecycle Management

6.1 Contract Definition

Contracts in Hyperweb follow a class-based pattern:

```
export default class Contract extends BaseContract {
    constructor(ctx: Context, state: State) {
        super(ctx);
        this.ctx = ctx;
        this.state = state;
    }
}
```

Listing 2: Base Pattern

6.2 Decorator Usage

The decorator system provides crucial contract lifecycle and access control features:

- Initialization control through `@instantiate`
- Access management via `@onlyOwner`
- Custom decorator support for extended functionality

7 HVM

The Hyperweb protocol introduces a novel approach to achieving genuine write-once, run-everywhere semantics for smart contracts and decentralized applications. At its core, Hyperweb defines a JavaScript-based execution environment, *HVM*, which operates natively on the Hyperweb chain. This enhanced VM supports serialization of the entire execution context, enabling the preservation and restoration of the environment at arbitrary points in the future. On external blockchains, an additional execution engine, referred to as an *interpreter*, interprets the same JavaScript code but is tailored to that respective chain's runtime (e.g., an EVM-based chain, CosmWasm, a Solana environment, etc). By unifying these execution contexts, developers write their logic once (in TypeScript, compiled down to JavaScript), and then seamlessly execute their code across a variety of heterogeneous chains without manual adaptation. This allows for several methods of computing using different variations of transmitted substate, and code.

To reason about the correctness, performance, and security properties of Hyperweb's universal VM abstraction, consider that each blockchain's state transition function can be modeled as:

$$f : S \times I \to S$$

where S is the set of all possible states of a particular chain, and I represents the input (transactions, messages, or other stimuli). In a traditional single-chain setup, f is fixed to a single VM or interpreter, but here we instead conceive a global transition function:

$$F : S_1 \times S_2 \times \cdots \times S_n \times I \to S_1 \times S_2 \times \cdots \times S_n$$

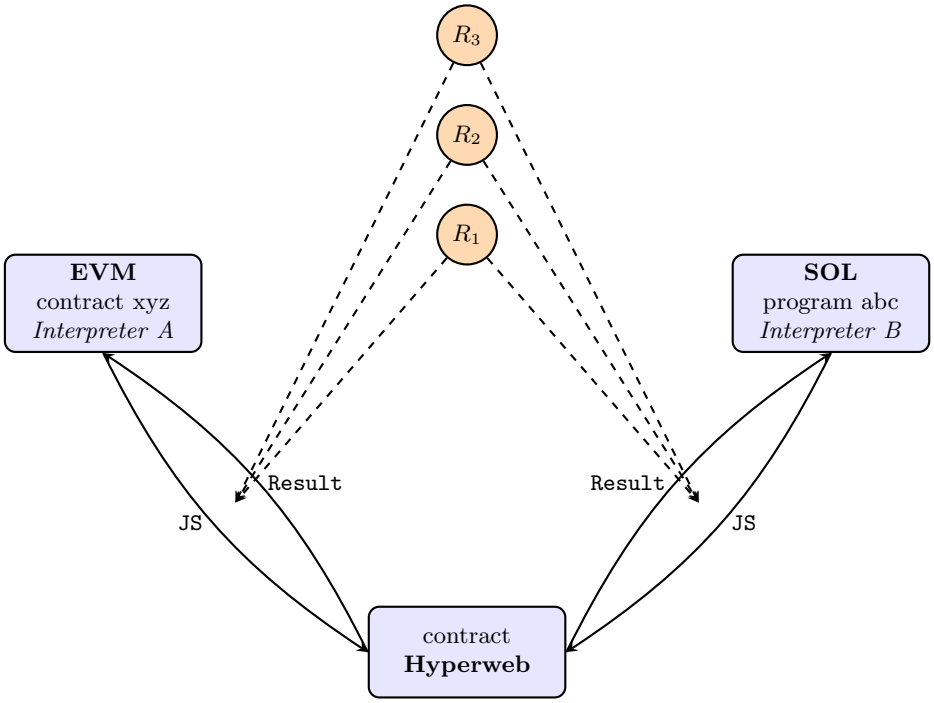


Figure 3: Hyperweb architecture diagram: multiple chains (EVM, SOL) each have their own interpreter and can exchange JavaScript calls and results with Hyperweb. Relayers R_1, R_2, R_3 facilitate communication between Hyperweb and the external chains by connecting to the communication channels between Hyperweb and each chain. A single TypeScript/JavaScript contract (MyContract) can coordinate logic across all connected chains.

for n distinct chains, each potentially running a different VM environment. Hyperweb integrates these states and interpreters into a unified logical environment. Essentially, F factors through a per-chain interpreter—HVM for the Hyperweb chain, and an interpreter for each foreign chain—and ensures that the same codebase drives state transitions everywhere.

A key challenge that arises is handling asynchronous cross-chain invocations. If a contract on the Hyperweb chain needs data from a contract deployed on another chain, it can issue a cross-chain call and eventually await its result. However, blockchains generally operate in a synchronous manner: a transaction must be fully determined at the time of execution. To reconcile these paradigms, we model cross-chain calls as deferred computations. We can represent a cross-chain call as a promise-like construct:

$$P : X \to Y$$

where X is the initiating chain's local request and Y is the eventual result resolved by the target chain.

This asynchronous abstraction means that a cross-chain promise may resolve at some future point in time. Execution that involves waiting on data from another chain will effectively pause until the promise resolves. To maintain safe execution semantics, Hyperweb enforces that any

cross-chain promise must be settled before the state transition is finalized. HVM achieves this by leveraging its serialization capabilities to suspend the current execution context when a promise is issued:

$$\texttt{Suspend}(\mathcal{E}_{\text{Context}}) \rightarrow \sigma$$

where, $\mathcal{E}_{\text{Context}}$ is the execution context. Once the promise P_{ij} is resolved, HVM restores the execution context from the snapshot, σ and resumes execution seamlessly:

$$\texttt{Resume}(\sigma)$$

This ensures that all asynchronous operations are handled gracefully, with execution contexts preserved accurately across suspension and resumption points. Consequently, the final state transition $F'(S_1, \ldots, S_n, I)$ incorporates the results of all resolved promises, maintaining consistency and reliability in the state management process.

We may represent the combined state transition after all cross-chain dependencies have resolved as:

$$F'(S_1, \ldots, S_n, I) = G(S_1, \ldots, S_n, I)$$

where G denotes the state transitions once all cross-chain calls have completed. this ensures that finality is only reached after the required asynchronous computations are concluded.

in essence, the universal VM (formed by the interplay of HVM on Hyperweb and interpreters on other chains) provides a uniform interface for:

1. state definitions using a consistent schema (e.g., Item, Map).
2. event emission that is chain-agnostic, ensuring identical developer experience and tooling compatibility.
3. use of decorators allow developers to specify conditions such as `@instantiate` or `@onlyOwner`

Mathematically, the advantage of this approach is a reduction in complexity. rather than reasoning about a separate VM f_i for each chain c_i, the developer can now think of a single global interpreter. we let:

$$\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$$

represent the set of interpreters, where each I_j is an interpreter instance running on a distinct chain, and define:

$$M = (H, \mathcal{I}) \quad \text{where} \quad H = \texttt{HVM}.$$

Where, M represents the meta-machine that combines the HVM (H) and the distributed set of interpreters ($\mathcal{I}$). This structure allows the execution semantics to be unified across chains while maintaining flexibility and chain-specific adaptations through the interpreters.

The execution semantics can thus be unified into a single abstract machine model:

$$f_{unified} : (S_1 \times \cdots \times S_m) \times I \rightarrow (S_1 \times \cdots \times S_m)$$

with the guarantee that all logic, state manipulation, and event emission follow a single coherent specification, just translated across multiple low-level runtimes.

This framework around asynchronous calls, unified state transitions, and a single codebase interpreted in multiple environments demonstrates a protocol that reduces the developer's mental overhead, simplifies the cross-chain lifecycle, and enables powerful features like asynchronous callbacks and secure, capability-based access control.

7.0.1 Extended State and Input Spaces

As previously mentioned, each chain c_i in a collection of n heterogeneous blockchains has a state space S_i. we can consider the product space:

$$\mathcal{S} = S_1 \times S_2 \times \cdots \times S_n$$

as the global state encompassing all chains simultaneously. likewise, let $\mathcal{I}$ represent the global input space, where an input $I \in \mathcal{I}$ may include operations to be applied across multiple chains.

The universal transition function was defined as:

$$F : \mathcal{S} \times \mathcal{I} \to \mathcal{S}.$$

however, this function F should be viewed not as monolithic, but as composed of individual per-chain functions coupled by cross-chain communications. formally, we can write:

$$F(S_1, \ldots, S_n, I) = (f_1(S_1, I_1), f_2(S_2, I_2), \ldots, f_n(S_n, I_n))$$

where each f_i is the local state transition function for chain c_i, and I_i is the subset of inputs relevant to that chain. the difficulty arises when these I_i and consequently f_i become dependent on outputs from other chains.

7.0.2 Asynchronous Behavior

To handle asynchronous calls, consider a scenario where chain c_i issues a request that must be fulfilled by chain c_j. Instead of f_i producing a final state immediately, it produces a *deferred state update* or a promise-like variable. We may represent this deferred outcome as:

$$f_i(S_i, I_i) = S_i' \oplus P_{ij}$$

where P_{ij} denotes a promise awaiting resolution from chain c_j. The operator $\oplus$ indicates that the resulting state S_i' is not final; it is augmented with a placeholder for future data.

The enhanced runtime environment incorporates robust snapshotting capabilities, allowing the entire execution context—including the call stack, variable states, and pending promises—to be serialized at the point where P_{ij} is issued. This serialized state can later be deserialized to seamlessly restore execution, ensuring that promises are resolved correctly without loss of context. This mechanism guarantees that execution involving waiting on data from another chain can be paused and resumed safely, maintaining consistent and secure execution semantics. This allows hyperweb to record the environment at the point where P_{ij} is issued and restore it once the promise is resolved, ensuring seamless continuation of execution.

chain c_j upon processing its own transitions, eventually resolves P_{ij}. once the promise P_{ij} is resolved and all other cross-chain dependencies have completed, we can define:

$$F'(S_1, \ldots, S_n, I) = G(S_1, \ldots, S_n, I)$$

where G represents the state transitions after all promises are resolved. this ensures that the final state is determined only after the required asynchronous computations are concluded.

7.0.3 Cross-Chain Code and State Distribution

See fig 4, beyond the baseline model of remote invocation, hyperweb defines a systematic approach for distributing code and state across multiple chains. let $\mathcal{C}$ represent the set of available code modules. consider a source chain c_i with state S_i, and a target chain c_j with interpreter $\mathcal{I}_j$ and state S_j. let $S' \subseteq S_i$ denote a chosen substate of the source chain's state.

hyperweb supports three canonical modes of code and state distribution:

1. **code-only execution:** given a code subset $\mathcal{C}' \subseteq \mathcal{C}$ and a target chain c_j with local state S_j, hyperweb executes $\mathcal{C}'$ against S_j without transferring any additional state. formally:

$$\Delta_{code} : (\mathcal{C}', c_j) \mapsto (S_j \xrightarrow{\mathcal{C}'} \hat{S}_j)$$

   where $\hat{S}_j$ represents the state of c_j after executing $\mathcal{C}'$ on $\mathcal{I}_j$.

2. **combined code and substate bundle:** for operations requiring a specific subset of state, hyperweb constructs a bundle $(\mathcal{C}', S')$ where $S' \subseteq S_i$. this bundle is shipped to c_j. the interpreter $\mathcal{I}_j$ then applies $\mathcal{C}'$ to $(S_j \cup S')$. formally:

$$\Delta_{bundle} : (\mathcal{C}', S', c_j) \mapsto ((S_j \cup S') \xrightarrow{\mathcal{C}'} \hat{S}'_j)$$

   this ensures that all necessary data is co-located with the code for that invocation.

3. **Contract Deployment:** Rather than a one-off execution, Hyperweb allows for deploying contracts permanently or temporarily onto target chains by leveraging the interpreter's contract factory and proxy capabilities. Specifically, the deployment process involves a deployment operation Δ_{deploy} that utilizes the interpreter's contract factory to install $(\mathcal{C}', S')$ onto c_j. This installation can generate either a persistent or temporary child contract (or a new account in account-based chains) through the interpreter's factory and proxy mechanisms, thereby serving as the runtime or execution environment for your contract's logic. Additionally, to manage the registry of active contracts, users can pay rent to the interpreter contract to maintain their contracts within its registry over time. Formally, this is represented as:

$$\Delta_{deploy} : (\mathcal{C}', S', c_j) \mapsto (\mathcal{I}_j^*, \hat{S}''_j, R_j)$$

   where:

   - $\mathcal{I}_j^*$ signifies that the interpreter at c_j now hosts the deployed contract—either as a persistent runtime or a temporary execution environment—and its associated state.
   - $\hat{S}''_j$ represents the updated state of chain c_j after deployment.
   - R_j denotes the rental agreement or registry entry for the deployed contract, ensuring it remains active within the interpreter's registry.

This allows for the flexibility of incoming executions to share global ephemeral, temporary state, or to maintain persistent state. Remote executions are initially restricted to be pure functions, whereby a subset of the function signature's parameters may be resourced defined to be available on the destination chain. Although importing modules is an open research question.

From an object capabilities perspective [8], each of these distribution mechanisms can be viewed as conferring specific, fine-grained rights to code and data. Let $\mathcal{C}$ denote the set of all possible capabilities, where each capability $c \in \mathcal{C}$ defines the right to access or modify specific resources or execute specific actions. These mechanisms are described as follows:

1. **Code-Only Execution:** Grants the recipient interpreter the capability $c_{\text{exec}} \in \mathcal{C}$ to perform a known set of computations against its local state S_j. Formally:

$$c_{\text{exec}} : S_j \to S'_j$$

where S'_j represents the state of the target chain c_j after executing the granted capability.

2. **Bundling Code with Substate:** Transfers a combined capability $c_{\text{bundle}} = (\mathcal{C}', S')$ where $\mathcal{C}' \subseteq \mathcal{C}$ and $S' \subseteq S_i$, granting access to a well-defined portion of the source chain's state. Formally:

$$c_{\text{bundle}} : (S_j \cup S') \to S''_j$$

ensuring that the invoked logic operates with the authority it needs.

3. **Contract Deployment:** Permanently bestows the remote interpreter with a persistent capability $c_{\text{deploy}} \in \mathcal{C}$ to host and execute the deployed contract's logic and state:

$$c_{\text{deploy}} : S_j \to (\mathcal{I}^*_j, S''_j)$$

where $\mathcal{I}^*_j$ indicates the interpreter $\mathcal{I}_j$ now persistently hosts the deployed contract.

7.0.4 Object Capabilities Security Model

This approach models the object capability security model [8], which directly contrasts with identity-based access control (IBAC), where rights are tied to an actor's identity. Instead, authorization-based access control (ABAC), as implemented with object capabilities, decouples authority from identity, leading to several measurable security advantages:

- **Elimination of Identity Verification Complexity:** In IBAC, access control depends on verifying the identity of the sender:

$$\texttt{verify}(id) : \texttt{Actor} \to \texttt{Boolean}$$

and mapping it to permissions. This introduces risks of spoofing and privilege escalation. In contrast, ABAC relies on direct possession of capabilities ($c \in \mathcal{C}$), which are unforgeable and self-contained.

- **Minimized Authority Leakage:** Each capability explicitly encodes the allowable actions, ensuring the principle of least authority (PoLA):

$$\forall c \in \mathcal{C}, \text{scope}(c) \subseteq R$$

where $\mathcal{C}$ is the set of capabilities, $\text{scope}(c)$ denotes the set of operations authorized by capability c, and R represents the resources required for execution.

In IBAC, broad identity-based permissions often lead to over-privileged actors.

- **Formal Measurability:** Capabilities provide a mathematically auditable chain of delegation:

$$c_{\text{deleg}} : \mathcal{C}_{\text{parent}} \to \mathcal{C}_{\text{child}}.$$

This ensures that authority transfer is explicit and traceable, unlike IBAC, where auditing identity mappings can be opaque.

By modeling cross-chain interactions through capabilities, Hyperweb ensures secure and efficient execution semantics. For example, a cross-chain invocation using a capability c_x is defined as:

$$c_x : (S_i, S_j) \to S'_j$$

where:

- S_i: The state of the source chain c_i, representing the initial conditions and data involved in the invocation.
- S_j: The current state of the target chain c_j, prior to the execution of the capability c_x.
- S'_j: The resulting state of the target chain c_j after executing c_x. This state reflects the updates applied through the invoked logic.

In this framework, c_x operates within explicitly defined bounds, ensuring that only authorized state transitions are performed. This approach eliminates the need for runtime identity checks or additional trust assumptions, as authority is encapsulated directly in the capability.

Capabilities also adhere to the principle of least authority (`PoLA`), ensuring that their scope is limited to necessary resources:

$$\forall c \in \mathcal{C}, \texttt{scope}(c) \subseteq R_{\text{req}}.$$

Where:

- $c \in \mathcal{C}$: Represents a specific capability from the set of all capabilities.
- $\texttt{scope}(c)$: The set of actions and resources explicitly defined as accessible by the capability c.
- R_{req}: The minimal set of resources necessary to perform the intended operation.

By adhering to these formal constraints, Hyperweb's capability-based approach ensures that authority is explicitly defined, reducing security risks and operational overhead.

7.0.5 State Modification

Understanding which components of the system's state are modified during contract deployment and execution is crucial for maintaining consistency and security across chains. Specifically:

- **Contract State S':** This represents the subset of the source chain's state essential for the contract's operation. When deploying a contract, S' may include configurations, permissions, or initial data required by the contract logic.
- **Interpreter State $\mathcal{I}_j$:** Deploying a contract modifies the interpreter's state on the target chain c_j. Specifically, $\mathcal{I}_j^*$ denotes the updated state of the interpreter after hosting the new contract, ensuring it can manage and execute the contract's logic consistently.
- **Global State $\mathcal{S}$:** The deployment operation Δ_{deploy} contributes to the global state transition by integrating the new contract into the ecosystem, thereby altering the collective state $\mathcal{S} = S_1 \times S_2 \times \cdots \times S_n$ across all chains.

This delineation of state modifications ensures that deployments are executed with precision and that each interpreter maintains an accurate and secure execution environment for the contracts it manages.

Combined with the enhanced snapshotting capabilities of the underlying runtime, these distribution mechanisms allow Hyperweb to serialize and restore execution environments at well-defined checkpoints. This capability enables more sophisticated cross-chain interactions, such as:

1. **Replay of Computations**: By restoring from a snapshot, Hyperweb can replay computations to ensure consistency and correctness across chains.
2. **Multi-Step Workflows**: Complex workflows involving multiple cross-chain calls can be managed by serializing the execution state at each step, allowing for precise control and recovery.
3. **Fault Tolerance**: In the event of failures or interruptions, the system can restore the execution context from the last snapshot, minimizing disruption and ensuring reliable operation.

Hyperweb can record the environment at the point where a promise is issued, and subsequently restore it once cross-chain dependencies are resolved. this facilitates scenarios such as replaying computations from known snapshots or orchestrating multi-step workflows across multiple chains, all while preserving the object-capability discipline.

7.0.6 Execution Context Serialization and Restoration

A pivotal enhancement in *HVM* is its ability to serialize and deserialize the entire execution context. This includes the current state of the call stack, local and global variables, and the status of all pending promises. When a cross-chain promise P_{ij} is issued, HVM captures a snapshot of the execution environment:

$$\texttt{Snapshot} = \texttt{Serialize}(\mathcal{E}_{\text{Context}})$$

This snapshot is stored securely and can be transmitted or stored as needed. Once the promise P_{ij} is resolved by chain c_j, HVM can restore the execution context from the snapshot:

$$\mathcal{E}_{\text{Context}} = \texttt{Deserialize}(\text{Snapshot})$$

This restoration ensures that the execution can continue precisely from where it was paused, with all variables and states intact. This capability is fundamental for maintaining the integrity of asynchronous operations and ensuring that the system can recover gracefully from interruptions or delays in cross-chain communications.

7.1 Computational Metering

The chain's VM uses a computational metering mechanism to ensure predictable execution limits for smart contracts. Metering is achieved via an interrupt handler implemented in the underlying runtime. This handler enforces an upper bound on execution steps, represented as `maxSteps`, which can be dynamically configured.

7.1.1 Step Counting and Interrupt Handling

The metering process integrates a step counter into the runtime. Each instruction executed by the VM increments the counter. When the counter exceeds `maxSteps`, the runtime triggers an interrupt, terminating the execution. Let S denote the current step count:

$$S = S_0 + \sum_{i=1}^{n} \delta_i,$$

where δ_i represents the step increment for the i-th instruction. The interrupt condition is defined as:

$$S > \texttt{maxSteps} \implies \texttt{terminate execution}.$$

7.1.2 Contract Execution with Metering

To demonstrate the metering system, we tested the HVM with both finite and infinite loop constructs in a sample contract. The state transition function f for metered execution is defined as:

$$f(S, I) = \begin{cases} S', & \text{if } S + \delta \leq \texttt{maxSteps}, \\ \text{error}, & \text{otherwise.} \end{cases}$$

For example, consider the following TypeScript contract with a finite loop and an infinite loop:

```
export default class MeteringContractTest {
  testWhileLoop(limit) {
    let count = 0;
    let sum = 0;
    while (count  limit) {
      sum += count;
      count++;
    }
    return sum;
  }

  testInfiniteLoop() {
    let count = 0;
    while (true) {
      count++;
    }
  }
}
```

Listing 3: Metering Contract

When executing the finite loop function `testWhileLoop`, the step counter increments until the specified limit is reached. For `limit` $= 10$, the expected sum `result` is computed as:

$$\texttt{result} = \sum_{i=0}^{9} i = 45.$$

For the infinite loop function `testInfiniteLoop`, the step count eventually exceeds `maxSteps`, triggering the interrupt handler. The interrupt handler guarantees that runaway computations are safely terminated, preserving the integrity of the blockchain state and preventing denial-of-service attacks.

7.1.3 Dynamic Metering Policies

The `maxSteps` value can be:

- Static: Hardcoded as a constant (e.g., `1000`).
- Dynamic: Configured based on gas supplied by the user.
- Governance-Driven: Determined via on-chain governance.

In the future, we may make enhancements to integrate dynamic gas pricing models, adjusting `maxSteps` proportionally to the gas supplied. The relationship can be expressed as:

$$\texttt{maxSteps} = \alpha \cdot \texttt{gas},$$

where α is a scaling factor defined by system parameters.

Figure 4: Sequence diagram illustrating HVM system's interaction with EVM, Solana, and CosmWasm blockchains. It demonstrates how the Hyperweb contract manages remote calls to interpreters on each chain and merges the results into a unified response.

8 Conclusion

The fragmentation of blockchain development has created a barrier to mainstream adoption, restricting innovation and limiting developer participation. Hyperweb represents a significant step forward in software development, providing a robust and developer-friendly platform for building a new wave of applications. By serving as the Interchain JavaScript Hub, Hyperweb acts as a unifying layer that seamlessly bridges Web2 and Web3, bringing developers together to build the next generation of the Web and the Internet as an open, interoperable ecosystem. By abstracting blockchain complexity and integrating familiar development paradigms, Hyperweb significantly lowers the entry threshold for millions of developers. This approach not only expands the reach of decentralized applications but also catalyzes a new era of scalable, user-centric innovation and use cases. The mass adoption of blockchain depends on an untapped source of developers who have yet to enter the space. The Web's evolution continues only when developers of all backgrounds can seamlessly contribute to both decentralized and traditional applications.

9 Bibliography

References

[1] S. Overflow, "Most popular technologies," 2020, released: 2020. [Online]. Available: https://insights.stackoverflow.com/survey/2020#most-popular-technologies

[2] Hyperweb, "Chain registry," 2025, accessed: 2025-01-30. [Online]. Available: https://github.com/hyperweb-io/chain-registry

[3] Supabase, "Supabase general availability," 2025, accessed: 2025-01-30. [Online]. Available: https://supabase.com/ga

[4] Hyperweb, "Hyperweb download count," 2025, accessed: 2025-01-30. [Online]. Available: https://github.com/hyperweb-io/lib-count

[5] E. Brewer, "Towards robust distributed systems (podc keynote)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, Portland, OR, USA, July 2000, co-Founder & Chief Scientist, Inktomi; Professor, UC Berkeley. [Online]. Available: https://sites.cs.ucsb.edu/~rich/class/cs293b-cloud/papers/Brewer_podc_keynote_2000.pdf

[6] E. Capital, "Electric capital developer report," 2024, accessed: 2024-12-26. [Online]. Available: https://www.developerreport.com/

[7] S. R. Department, "Worldwide developer population 2023," 2023, released: August 2023. [Online]. Available: https://www.statista.com/statistics/627312/worldwide-developer-population/

[8] M. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," 2006, chief Scientist of Agoric Systems. [Online]. Available: https://jscholarship.library.jhu.edu