# ChainUI: A Decentralized Deployment Framework for Component-Based Web Applications

Grzegorz Raczek
grzegorz@vestige.fi
www.chainui.com

July 3, 2023

**Abstract**

This paper presents ChainUI, a novel approach to the durability problem of Web3 interfaces, addressing security risks, app versioning and component redundancy. It assesses the current solutions available and proposes a novel method using smart contract storage. By deploying web application code into application and component smart contracts, ChainUI makes the interfaces as enduring as the underlying network, overcoming the limitations of memory and ensuring regular content updates. Through these innovations, ChainUI addresses the major concerns surrounding Web3 interfaces, paving the way for a more robust, resilient, and efficient framework for decentralized web applications.

# Contents

# 1 Introduction

Web3 applications, in their current state, have predominantly adopted the deployment strategies of their Web2 predecessors, with minor adaptations to accommodate Web3 functionalities such as wallet connectors and, in certain cases, network configurations. These adjustments have been pivotal in rendering smart contracts user-friendly and operational for those who lack technical expertise. The problem lies in the fact that these practices are deeply rooted in centralization, mirroring the architecture of the Web2 era. Consequently, a vast majority of users still rely on centralized hosting solutions for fund management, paradoxically counteracting the decentralization ethos that lies at the heart of blockchain technology. This whitepaper elucidates novel solutions to address this discrepancy, detailing how it's possible to leverage the blockchain's decentralized infrastructure for improved durability and user interface management.

# 2 The issues of Web3 interfaces

Despite the emergence of blockchain technology, the existing generation of web applications continues to largely rely on centralized infrastructures. This inherent centralization introduces a variety of vulnerabilities. These include a dependence on a single point of failure, additional security risks, the possibility of user data misuse and a lack of overall user control.

## 2.1 Security risks

The conventional architecture of web applications is predominantly centralized, resulting in a single point of failure. This means that if the central server is compromised or fails, the entire service becomes inaccessible. For Web3 applications, the consequence of this failure could be even more severe as users may potentially lose access to their funds. Centralized servers, cloud hosts, or Domain Name Servers (DNS) are prime targets for cyber-attacks, as disruption here cripples the entire application. This system not only presents a considerable risk to the availability and reliability of the application but also threatens the integrity of user data stored or processed through these servers. This vulnerability is well illustrated by JavaScript substitution attacks, wherein a server is manipulated into disseminating harmful JavaScript code instead of the legitimate one, jeopardizing user data and possibly causing considerable harm.

A practical example of this security flaw is the recent cyber-attack on MyAlgo Wallet, which was thoroughly documented in a Twitter thread by MyAlgo [1] and an incident report by Halborn Security [2]. In this case, the perpetrators capitalized on the centralized infrastructure, abusing Cloudflare's Content Delivery Network (CDN) to inject malicious code through a man-in-the-middle attack between the actual wallet web application and the user [1]. This occurred via a potentially compromised CDN API key, although the exact means of its acquisition remain unclear [1]. The MyAlgo codebase itself displayed no signs of exploitation or vulnerability, and there was no evidence to suggest that the CDN user account was compromised [1].

## 2.2 App versioning

The versioning system of most web applications also presents a substantial problem. In our rapidly evolving digital landscape, software is constantly updated and improved. It's crucial for users to be able to engage with different iterations of a program, not only for preference but also to ensure backward compatibility with previous smart contracts. Yet, under the prevailing model, older versions of applications are frequently overwritten and/or phased out.

This prevalent practice can lead to a multitude of problems. Users can experience a loss of functionalities previously available, or in more severe cases, lose access to funds locked in smart contracts that were supported by earlier application versions. This issue arises due to current development frameworks being designed with their transitory nature in mind. This inherent design perspective results in a natural misalignment of development cycles between smart contracts, which are immutable and persist over time, and user interfaces that are subject to continuous updates and changes.

## 2.3 Monolithic design

Monolithic design further exacerbates the issues with the centralization of web applications. In this design paradigm, the entire codebase of the web application is developed, maintained, and deployed as a single entity. This approach can lead to problems in terms of scalability, maintenance, and updates. If one feature requires updating or patching, the entire application must typically be redeployed.

Moreover, a monolithic design tends not to facilitate component redundancy or the ability to work independently. This results in the application being only as strong as its weakest component, and the failure of a single component could disrupt the functionality of the entire application. Furthermore, this design approach is less adaptive to changes in the ecosystem. In the blockchain world, a monolithic design can make the task of ensuring application longevity considerably more challenging.

This design pattern also poses difficulties when it comes to app versioning. Different versions of the application, in a monolithic setting, can be hard to manage and may not coexist easily, leading to the aforementioned issues of incompatibility with older smart contracts.

### 2.4 Implications

The challenges we've discussed can create significant obstacles to realizing a completely decentralized internet (Web3), limiting the potential of blockchain technology to fully revolutionize our online interactions and transactions. We need a solution that ensures the longevity of web applications by effectively managing multiple software versions, while also seamlessly merging smart contracts with their respective interfaces.

## 3 Available solutions

The challenges in achieving decentralized user interface storage are significant, but a number of emerging solutions are being explored within the blockchain industry.

### 3.1 Decentralized Storage Networks

One such approach employs Decentralized Storage Networks (DSNs). These systems, including IPFS[4] and Filecoin[5], use a network of nodes to store data. Leveraging cryptographic hashing and peer-to-peer protocols, DSNs ensure that data is stored securely, redundantly, and permanently across a distributed network. These systems are capable of storing static files, which may include the HTML, CSS, and JavaScript files required for a web interface.

While Decentralized Storage Networks (DSNs) offer several advantages, they still fall short in addressing certain challenges. For example, while they are adept at hosting static files for a web interface, they struggle with handling content updates. This is due to the immutable nature of data stored on such networks, where any alteration requires a completely new upload and corresponding hash address.

Furthermore, while DSNs like IPFS can promise content permanence, they do not guarantee data availability. This means that the availability of a webpage's user interface on IPFS relies heavily on the network's nodes continually deciding to host the content, something that cannot be assured indefinitely. Therefore, while the smart contracts may be operational 24/7 due to the blockchain's inherent design, this does not necessarily translate to the uptime of the user interface hosted on a DSN.

### 3.2 Decentralized Cloud Storage

Another promising alternative involves decentralized cloud storage platforms like Storj [6] and Sia [7]. These platforms function similarly to traditional cloud storage services, but instead of storing data in centralized data centers, they distribute data across a network of individual nodes. This ensures that the data remains accessible even if one or several nodes in the network go offline.

Decentralized cloud storage platforms are promising in ensuring data availability and redundancy. However, they also face challenges in effectively serving web applications. Similar to DSNs, the handling of dynamic content is a significant hurdle. As these platforms are designed for general file storage, they are not necessarily optimized for serving web content, which may lead to slower load times and inefficient data retrieval.

Additionally, these solutions do not inherently solve the app versioning problem. While they may store different versions of an application, they lack a built-in system to handle version management, leaving it up to individual users or developers to manually track and manage versions. This can lead to confusion and increases the risk of users interacting with outdated or incompatible versions of web applications.

Finally, these platforms still require integration with the traditional Domain Name System (DNS) for users to access the stored content easily. This reliance on a centralized system contradicts the goal of complete decentralization, reintroducing a single point of failure.

# 4 Proposed solution

The aim of this project is to establish a robust system that transcends the limitations of the current Web3 infrastructure by leveraging the inherent strengths of the blockchain. The proposed solution embeds web application code within smart contracts to ensure permanent availability. The web application can be constructed via a specially designed renderer which retrieves the code directly from the blockchain, making the website usable and its uptime congruent with the uptime of the entire network. ChainUI ensures that the user interface exhibits the same degree of resilience and durability as the underlying smart contracts. It also makes several improvements to the security and user experience of Web3 applications.

The framework is composed of three parts, all of which need to be network-specific:

1. **Smart contract storage** - meaning the smart contract code that will be deployed on the blockchain network. It will store application data through any means available (state, box storage, etc.);

2. **Code renderer** - meaning the rendering engine which will display the user interface after receiving a correct application reference (id, hash, etc.);

3. **Developer build tools** - meaning the command line interface and any other tools provided to the developers which will enable them to build and deploy a web application.

All three of these components deal with complex problems related to blockchain development, which will be discussed in detail below. The blockhain layer used for consideration of below limitations will be the Algorand MainNet Network [8].

## 4.1 Smart contract storage

The fundamental bedrock of the ChainUI framework lies in the smart contract (SC) storage. Historically, smart contracts have served a myriad of purposes, including complex computation execution, data storage, and enforcement of logical rules.

Utilizing smart contracts to store application code brings forth a novel set of challenges and considerations. To optimize the developer experience and cultivate a thriving ecosystem, ChainUI's smart contracts necessitate a bifurcated structure:

1. **Component Smart Contract** - This refers to a smart contract that preserves a specific, discrete segment of an application, analogous to the approach of component creation in frameworks like ReactJS [10]. These components are designed to utilize parameters relayed from the higher-level render. To prevent misuse and/or potential code plagiarism by other developers (such as redeploying an existing component, thereby severing the connection between the original component code and the application), these components should not incorporate any other components. They should also store their content hash with any duplicates disallowed.

2. **Application Smart Contract** - This is the principal smart contract code that maintains all core functionalities of an application, capable of using components to render certain parts of it. Upon the deployment of this smart contract, it should be obligatory to compensate all the components used within the application. To ensure compliance with this rule, references to the utilized components should be embedded at the top-level of the smart contract, providing easy accessibility. The code renderer should then only load data for components which have received their due payment. This policy ensures the establishment of an equitable ecosystem where developers are duly rewarded for their contributions whenever their work is included in user-facing applications.

### 4.1.1 Memory limit

A principal obstacle is the memory constraint enforced by the majority of blockchain networks on smart contracts. For instance, Algorand stipulates a maximum of 8KB for an application's global state [9] and a cap of 32KB for each box storage [9]. This allocation can rapidly deplete while storing application data, disregarding even the inclusion of multimedia content such as images or videos.

One can navigate around this restriction by confining the storage of indispensable application components within the contract. This includes elements like the primary logic and data structures, whilst deferring non-critical items like common framework libraries (wallet connectors, ReactJS [10], AlgoSDK) to an external domain. Further, UI assets like images and media can be relegated to decentralized storage networks such as IPFS. Consequently, even if these assets are absent from other decentralized storage avenues, they won't hinder the loading of the page's core functionality.

Another enhancement to this constraint lies in the conceptualization of standalone components as distinct component smart contracts, which proffers dual advantages. Firstly, it empowers other developers to leverage the same snippets of code, thereby streamlining the development process for all parties involved. Secondly, it provides a mechanism for the primary developer to monetize their contribution to the ecosystem. This model engenders a mutually beneficial situation: other developers gain ready access to high-quality components, reducing their workload, while the primary developer not only receives compensation for their efforts but also attains the capacity to store more data within a smart contract.

Even including these solutions, optimal data structuring, streamlining, and compression gain crucial importance. Developers need to consistently monitor the build size of their applications. Resources such as templates, code boilerplates, and an extensive set of best-practice examples should be readily accessible for anyone developing with ChainUI. While storing a single app across multiple smart contracts or boxes is feasible, it's not advisable due to the unnecessary complexity it introduces, thereby degrading the user experience.

### 4.1.2 Content updates

In most blockchain networks, smart contracts are intrinsically immutable. This means that once deployed, the coded instructions contained within them are unmodifiable. While this immutability confers a high degree of security and trustworthiness to the contract, it simultaneously introduces a considerable challenge when it comes to updating application functionality.

In contrast, updates to traditional web applications are relatively straightforward. Developers can easily deploy updates or patches to the server, instantly reflecting these changes to the user base. To replicate this streamlined update process for Web3 applications, it is essential to devise mechanisms that allow smart contract owners to modify the stored code.

Fortunately, the Algorand blockchain accommodates contract upgrades. Additionally, it enables updates to data stored in the global state or box state, thereby simplifying data modifications. This feature may not be available in other blockchain networks, which highlights the significance of facilitating content updates in smart contracts.

However, the mechanisms to perform updates should differ between the two types of smart contracts—application and component. Application smart contracts need to be effortlessly updatable and always present the most recent version to the user. In contrast, component smart contracts should be updated less frequently, with stringent version monitoring during their usage in application rendering. Provisions must be made to safeguard users, even in the event of potential threats from malicious actors.

### 4.1.3 Version management & content accessibility

The necessity of version management for the rendering of web applications cannot be understated, given its importance for ensuring security and facilitating effortless retrieval of historical versions. Current versions of the Algorand Indexer [11] are suitably equipped to provide this feature. Specifically, it allows data to be readily recovered and temporally shifted, contingent on the transaction history and state deltas of the associated smart contracts.

Nevertheless, the increasing scale of the project over time proportionately elevates the complexity of this issue. The ongoing growth of the blockchain necessitates an exhaustive indexer infrastructure, the implementation of which might be either unattainable or the resulting benefits might not offset the consequent expenditures in the frame of the ChainUI project. To simplify this procedure, utilization of a Conduit [12] might be a viable option, but the recommendation leans toward the employment of box storage instead.

Subsequent versions should be deployed as supplementary box storage of the application. Accessing alternate application versions can be then achieved by retrieving the specific storage box corresponding to that version, or by recalculating modifications based on the delta information stored within the box. This methodology greatly enhances the overall efficiency of the process. It necessitates only a single network node for the complete support of the ChainUI system, which can feasibly be managed locally using relatively low-cost hardware.

### 4.1.4 Deployment

The deployment strategy for smart contracts ought to be standardized and catalogued by a centralized management smart contract, whose role is to ensure payment verification for all parties, assess the validity of each smart contract, and manage the registration of these smart contracts for decentralized utilization. Provision should be made for templates of suitable smart contracts that can be readily incorporated into the developer tools required for constructing web applications.

A critical aspect of this process, particularly in the context of the application smart contract, entails the remuneration for usage directed towards other developers, and the accurate allocation of other smart contract references within the application's state. The central smart contract should enforce this payment protocol and prohibit registration within the project in the event of non-compliance. This procedure is to be reiterated for every smart contract update, although payment should be verified only upon addition of previously unused components.

Major revisions to the ChainUI project ought to incorporate new management contracts to ensure accurate interpretation of deployed application data by code renderers. This approach will enhance the long-term adaptability and maintainability of the system as well as continuous support of thepreviously deployed smart contracts.

## 4.2 Code renderer

The code renderer is a crucial part of the ChainUI framework, responsible for retrieving and rendering the web application code stored within the smart contracts. It is also the only part used by non-technical users, making it possibly the weakest link of the entire project. To ensure seamless operation and high-quality user experience, the code renderer must overcome several significant challenges related to decentralization, fetching the code, interpreting it correctly, and effectively abstracting transactions and wallet connectors.

### 4.2.1 Decentralization

Given that the entirety of the code renderer cannot be stored on the blockchain, it becomes crucial to ensure its availability via alternative avenues, such as IPFS [4], various storage providers, or significant simplification of the code renderer to an extent where it becomes exceedingly straightforward to recall. Ideally, most of the renderer's code should be embedded on-chain. This would enable users to write down or remember the code renderer application identifier and subsequently embed the code into an HTML iframe, thereby automating all other requisite operations.

The utilization of multiple, hard-to-control storage mediums is recommended in order to preclude third-party entities from erasing or modifying the content stored on the blockchain network. The priority should always be given to the renderer's availability over its functionality. This means that the integration of new features should be carefully considered; they should only be implemented if they do not undermine the ease of accessing the renderer.

### 4.2.2 Code fetching

The incorporation of diversified data retrieval avenues, preferably those under the complete ownership and management of application users (that is, self-run nodes), is of utmost significance. It is advised that maintainers of ChainUI supply a contingency data source, the operation of which could be financed by revenues generated through the project. Any additional publicly accessible data sources, including nodes managed by other developers within the ecosystem, should be integrated into the code renderer. The availability of a wider range of data sources enhances the overall reliability and efficiency of the system.

Ensuring the accuracy of the data is paramount and should be guaranteed through the application of cryptographic hash algorithms, thereby safeguarding against potential node manipulation and anticipated JavaScript substitution attacks. Every data retrieval operation should be accompanied by the confirmation of the data hash fetched from a separate, independent node. This hash ought to be ubiquitously embedded across the system, for instance, in the global state of the application as well as in the deployment transaction's note field. Prior to rendering the JavaScript code in the user's browser, the hash should be subjected to local verification to affirm its authenticity and proper retrieval.

### 4.2.3 Code interpretation

Following the assurance of code security, it becomes essential to decompress the code and render it using an appropriate framework in a uniform manner. For conventional JavaScript (JS) code, this might be as straightforward as integrating the code into a pre-existing HTML template. Considering that all references associated with other application identifiers are incorporated prior to the decompression of the code, it becomes feasible to retrieve them simultaneously, facilitating a relatively simultaneous display of the entire webpage.

The code renderer should be designed to remain neutral, neither intervening in nor modifying the presentation of content encapsulated within on-chain applications. It should not fall within the purview of the code renderer to amend missing references or similar errors. It is critical to ensure consistent functionality and display across various code renderers. As a result, these issues should be addressed during the build and deployment stages, rather than at the rendering stage, to prevent unintentional modifications to the intended behavior of the application as conceived by the developer. In instances where an application code proves unrenderable, the user should be notified of this circumstance without any further attempts to rectify these issues.

### 4.2.4 Abstraction of prevalent components

To enhance the functionality and efficiency of the ChainUI system, it is essential to incorporate the abstraction of frequently employed components within the code renderer. These components can be broadly classified into two categories:

1. **Network configuration**

   The network configuration is a crucial aspect in ensuring seamless user experiences in a Web3 environment. Aspects such as wallet connections, node configurations, and management of transaction messaging between the network and the application, can be entirely integrated into the code renderer. This comprehensive integration offers several advantages to both users and developers, including: streamlined node configurations, the requirement of a single wallet connection for the user to access all applications available on ChainUI, simplified transaction signing processes for developers, and a reduction in the build size.

   The components encompassed in the network configuration should manifest as visual elements, rendered as an unobtrusive, customizable overlay that appears upon hover, or through any other implementation that remains consistent across renders of different applications. These prepackaged functions can be made available to a rendered application through the use of the *window* [13] object or any other global reference method.

2. **Developer-requested imports**

   This category encapsulates the libraries that can be seamlessly located and embedded into the rendered application. It includes libraries such as AlgoSDK, ReactJS, and other popular libraries accessible via JavaScript Content Delivery Networks (CDNs) like cdnjs [14], Cloudflare JS CDN [15], or jsDelivr [16]. These libraries can then be embedded by also using the *window* object or any other viable method.

   The requested imports should be specified by the developer, ideally in the form of application state, in a manner that can be readily interpreted and fetched from any available CDN networks. The format of these requests should uniform and independent of any content delivery method used, such as the NPM[17] name and version of the library. This leverages the fact that removing these libraries from CDNs would significantly disrupt most of Web2. However, lesser-known libraries, although accessible from CDNs, should not be included due to the potential risk that their removal or inaccessibility could disrupt the functionality of the ChainUI app, while leaving most other websites unaffected.

In order to circumvent any unwarranted expansion of the code renderer's size, it is vital that it remains devoid of any extraneous libraries, save those pertinent to network connectivity. Crucially, the code renderer must abstain from imposing any global styling rules and any other modifications which could interfere with the usability of the rendered application.

## 4.3 Developer build tools

Developer tools play a pivotal role in fostering an efficient development experience while satisfying the prerequisites of the ChainUI system. Although application development should not deviate significantly from conventional standards, certain discrepancies might arise during later stages.

### 4.3.1 Development

To ensure a familiar experience akin to existing build tools, the implementation of ChainUI developer tools should primarily leverage a Command-Line Interface (CLI), entrusted with supervising all stages of development. These tools should capitalize on existing technologies such as Webpack [18] and package manager scripts defined in the *package.json* file.

For an intuitive and consistent work experience, the entire development cycle should employ a standard code renderer, mitigating potential frustrations in subsequent stages of application development.

### 4.3.2 Build output

The necessity of storing the entire application on-chain lends preference to housing it within a single data structure. This can be achieved using a standard HTML template. Unlike conventional development where JavaScript, CSS, and HTML files exist separately, they could be merged into a cohesive unit. This consolidation simplifies storage and facilitates more efficient optimization.

To ensure code comprehensibility, it would be beneficial to also deploy code source maps. However, since these aren't vital, they need not be stored on the node. They can conveniently be stored off-chain following their transmission in a transaction note or any other (preferably durable) means of storage.

### 4.3.3 Dependencies

The inclusion of library dependencies into the project without embedding them in the project's code constitutes a significant aspect of the process. As discussed earlier, some of these will be prepackaged into the code renderer, while others will be fetched as per the developer's request. These requests fall into two categories: libraries available on-chain such as component smart contracts, and libraries unavailable on-chain like AlgoSDK or ReactJS.

Although it might be feasible to incorporate these into the application's build file, many of these libraries have not been optimized sufficiently to be embedded without straining smart contract space. They can be stored and maintained by the developers of the ChainUI project on-chain or dynamically loaded from Content Delivery Networks (CDNs). Library names and versions should be archived in the smart contract metadata to enable the renderer to correctly interpret and display the web application.

Regarding components stored on-chain, the code renderer should provide functions to fetch and render these components, passing all available props. Upon building, references to these components should be extracted from the source code and then embedded in the smart contract metadata for accurate rendering.

### 4.3.4 Build size optimization

One of the most critical processes to prepare the web app for storage is optimizing the build file's size. This can be achieved using any combination of prevalent lossless compression algorithms such as gzip[19], Brotli[20], or lz-string[21]. The precise combination of these algorithms should be preserved in the smart contract metadata, allowing the code renderer to decompress and render the original file as intended.

Code renderer should provide all required libraries to decompress the stored on-chain data. As before, these libraries can also be stored on-chain or dynamically loaded from CDNs.

# 5 Conclusion

This paper introduced ChainUI, a novel decentralized deployment framework for component-based web applications, designed to address a number of key issues found in Web3 interfaces. Traditional Web3 interfaces are fraught with security risks and struggle with effective app versioning. These difficulties, while substantial, are not intractable, and we have examined existing solutions, such as decentralized storage networks and decentralized cloud storage, as pathways towards resolution.

However, these extant solutions are not sufficiently customised for user interfaces, which motivated us to design the ChainUI framework. In ChainUI, web application code is deployed into smart contracts, and a specialized renderer facilitates code loading directly from the blockchain. The system comprises two parts: an application smart contract and a component smart contract. Both types store code on-chain, with the former using the latter. Critically, the smart contracts incorporate a versioning system, enhancing code management and accessibility.

Nevertheless, we recognized that the adoption of such a system faces challenges related to smart contract memory limits, content updates, and deployment. To address these, we proposed a solution that makes use of a code renderer, which through code fetching and interpretation displays the intended application to the end user. To facilitate this system, we also proposed the creation of build tools to streamline the development process and optimize build size.

In conclusion, the ChainUI system offers a promising solution to address the persistent issues facing Web3 interfaces. By ensuring persistent availability and operational functionality of web applications, this solution represents a significant leap forward in the domain of decentralized applications (dApps). The versioning system provides an efficient way to manage application updates and to ensure the stability of older versions. Despite the potential challenges and constraints, we believe that the robustness and resiliency of the proposed system provide a viable solution to persistent issues in the field. Future work may extend the ChainUI framework to include additional features and improvements to ensure the system's viability and utility in an ever-evolving technological landscape.

## 6 Acknowlegements

I would like to express my deepest gratitude to Erik Hasselwander and Mariano Dominguez for their invaluable contributions to our discussions. Erik's innovative suggestion to employ a box storage-based versioning system, rather than relying on transaction delta recovery, significantly enhanced the efficacy and self-reliance of the project by eliminating the need for indexers and other sources of history management. Mariano's diligent proofreading and insightful comments played an instrumental role in refining the final version of the paper. Their collective expertise and dedication have undoubtedly enriched this project and the research experience.

## References


[1] MyAlgo. *Final Hack Findings and Report.* Twitter, April 21, 2023

`https://twitter.com/myalgo_/status/1649427794139525121`

[2] Halborn. *RandLabs - MyAlgo Wallet, Executive Summary.* GitHub, April 19, 2023

`https://github.com/HalbornSecurity/PublicReports/blob/04eea6a271aac7457a325e4decc7c2ac55e7ff5d/Incident%20Reports/RandLabs_MyAlgo_Wallet_Executive_Summary_Halborn%20.pdf`

[3] js-algorand-sdk. *The official JavaScript SDK for Algorand.* GitHub. Accessed July 3, 2023

`https://algorand.github.io/js-algorand-sdk/`

[4] InterPlanetary File System (IPFS). *IPFS Powers the Distributed Web.* Accessed July 3, 2023

`https://ipfs.tech/`

[5] Filecoin. *A decentralized storage network for humanity's most important information | Filecoin.* Accessed July 3, 2023

`https://filecoin.io/`

[6] Storj. *Storj - Make the world your data center.* Accessed July 3, 2023

`https://www.storj.io/`

[7] Sia. *Sia - Decentralized data storage.* Accessed July 3, 2023

`https://sia.tech/`

[8] Algorand. *Algorand | The Blockchain for FutureFi.* Accessed July 3, 2023

`https://algorand.com/`

[9] Algorand Developer Portal. *Contract storage.* Accessed July 3, 2023

`https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/state/`

[10] React. *The library for web and native user interfaces.* Accessed July 3, 2023

`https://react.dev/`

[11] Algorand Developer Portal. *Indexer*. Accessed July 3, 2023

`https://developer.algorand.org/docs/get-details/indexer/`

[12] conduit. *Algorand's data pipeline framework.* GitHub. Accessed July 3, 2023

`https://github.com/algorand/conduit`

[13] W3Schools. *The Window Object*. Accessed July 3, 2023

`https://www.w3schools.com/jsref/obj_window.asp`

[14] cdnjs. *cdnjs - The #1 free and open source CDN built to make life easier for developers*. Accessed July 3, 2023

`https://cdnjs.com/`

[15] Cloudflare. *cdnjs.cloudflare.com*. Accessed July 3, 2023

`https://cdnjs.cloudflare.com/`

[16] jsDelivr. *jsDelivr - A free, fast, and reliable CDN for JS and open source*. Accessed July 3, 2023

`https://www.jsdelivr.com/`

[17] npm.js. *npm*. Accessed July 3, 2023

`https://www.npmjs.com/`

[18] webpack. *webpack*. Accessed July 3, 2023

`https://webpack.js.org/`

[19] GNU Project. *Gzip - GNU Project - Free Software Foundation*. Accessed June 22, 2023

`https://www.gnu.org/software/gzip/`

[20] Google. *Brotli compression format*. GitHub. Accessed June 22, 2023

`https://github.com/google/brotli`

[21] pieroxy. *LZ-based compression algorithm for JavaScript*. GitHub. Accessed June 22, 2023

`https://github.com/pieroxy/lz-string`