Blockless Protocol / Overview

Blockless is the platform to launch, secure, and integrate Network Neutral Applications (nnApps) at unprecedented speeds.

Start building on Blockless Network today.

Vision

Welcome to the New Modular with Blockless.

One of Bitcoin’s biggest selling points was removing the need for and reliance on banks and other centralized institutions that hold your money, print more of your money, and restrict your access to your money.

We believe that the next challenge is removing the need for and reliance on networks that host our applications, limit our applications, and restrict our applications' performance, usability and potential to scale. Applications should be “Network Neutral”.

If modular blockchain architecture is taking networks apart and allowing blockchain to do what it does best, with other layers supporting for execution…

The modular application architecture with Blockless allows for decentralized applications to operate outside of the constraints of blockchain networks and make their own decisions about workloads and consensus - decisions previously out of their hands.

This means that applications no longer have to sacrifice performance and usability to fit within the capabilities of their blockchain network, and for the first time, full-stack decentralization becomes possible - no matter the application, no matter the network.

Technical Features

Automated Orchestration

Workloads deployed on Blockless are matched and distributed to most the suitable nodes with pseudo-randomness via our advanced selection and distribution algorithms, ensuring the most efficient execution for your application.

Dynamic Consensus and Verification

Select the ideal consensus mechanism for your nnApp, with customizable options such as data aggregation, pBFT, or RAFT.

Blockless further enhances security and transparency by offering optional computation verifiability through zero-knowledge proofs.

Other Notable Features

Ultra-portable node software: Utilizing WebAssembly (WASM) and the WebAssembly System Interface (WASI), Blockless enables communities to use a wide range of personal computers (including laptops and phones) to empower network neutral applications and a new modular paradigm for Web3.

Fully extendable runtime: Blockless dynamically allocates network resources to accommodate fluctuating workloads of nnApps, allowing developers to focus on building innovative applications instead of manually managing the performance and uptime of their specialized networks.

Home
Protocol
Core Concepts
Core Concepts
Modular application architecture

Blockless aims to pioneer the modular application paradigm in Web3. This architecture lets Web3 apps run on specialized networks for specific workloads, tapping into blockchain primarily for authentication, value transfer, and

ensuring system trust and integrity. It offers a balanced mix of blockchain benefits and performance optimization while preserving decentralization.

Network Neutral
Being network neutral means to not be limited or constrained by any particular infrastructure networks. Decentralized applications are often limited by the type of workload the underlying blockchain can support, and also the amount of work the public network can process at a given time. This hinders their ability to efficiently manage tasks like intensive computations, real-time processing and collaboration.

Network Neutral Applications (nnApps)
nnApps are decentralized applications that work across any L1/L2 blockchain network without being constrained by any particular blockchain's limitations such as latency, cost, and smart contract capabilities.

They leverage a new modular application architecture that we've built to improve security and performance. Our dynamic consensus mechanism forms part of this, allowing for applications to select the most efficient consensus and/or verification algorithms for each and every workload, which was previously impossible. Our automated node orchestration forms another part of our modular application architecture, allowing for the best-suited nodes for each and every workload to be selected at random to complete any specific task.

Specialized network
In the Blockless ecosystem, specialized networks are application-specific networks, spawned by the user, that offer a trustless and decentralized environment for executing tasks beyond the scope of smart contracts. These networks enhance the versatility and functionality of smart contract based applications, all while preserving their inherent trust and verifiability.

WASM
WebAssembly, often abbreviated as WASM, is a low-level, binary instruction format designed for safe and efficient execution. As a compact binary format, WASM enables faster loading, parsing, and execution times compared to traditional JavaScript. It is designed as a portable target for the compilation of high-level languages like C, C++, and Rust, allowing developers to run their code at near-native performance across different platforms (CPU architecture, OS, etc.).

In the context of Blockless, WASM is a major component of our service offering. It allows for the efficient execution of complex applications in a decentralized and trustless manner. By leveraging the advantages of WASM, Blockless provides a fast, secure, and scalable platform for running a wide range of applications.

WASI
WebAssembly System Interface, or WASI, is a system interface designed to work with WebAssembly modules. WASI provides a set of standardized APIs that allow WebAssembly applications to interact with the underlying operating system in a secure and platform-independent manner. This abstraction layer enables developers to write and run their applications on various systems without worrying about platform-specific details (type of CPU, OS, etc.).

In Blockless, WASI plays a crucial role in ensuring the seamless operation of nnApps. By providing a standardized interface for WebAssembly modules, WASI enables the interoperability of applications across different nodes in the Blockless network. Additionally, it ensures a secure execution environment by limiting access to system resources and enforcing strict security policies.

Blockless Runtime
The Blockless Runtime is a lightweight and efficient execution environment built on WebAssembly (WASM) technology, designed for secure and scalable serverless computing. It supports various programming languages, including C/C++, Rust, Swift, AssemblyScript, and Kotlin, and can run on any device with browser

support. The runtime incorporates features such as an optimizing code generator, low-overhead transitions, and the WebAssembly System Interface (WASI) for standardized interaction with the host operating system.

To ensure the security of its sandboxed environment, Blockless Runtime employs multiple mechanisms, including WebAssembly's inherent design, linear memory and bounds checking, validation, a capability-based security model, and sandboxing. Additional security measures such as JIT compilation with Control Flow Guard (CFG) and continuous updates from the open-source community further bolster its reliability.

Fault Tolerance
Fault tolerance is the ability of a system to continue operating correctly and maintain its functionality in the presence of failures, errors, or disruptions. In the context of operating a specialized network for a nnApp, fault tolerance ensures that the applications remain available and responsive even when individual components, nodes, or resources within the system experience failures.

In Blockless, this is achieved through redundancy, replication, load balancing, and automatic recovery, which collectively contribute to the resilience and reliability of the system.

Distributed Orchestration
Distributed orchestration is the process of coordinating and managing the execution of tasks across multiple nodes or servers in a distributed system. In Blockless, distributed orchestration mechanisms are responsible for efficiently allocating and managing resources, executing tasks, and ensuring nnApp workloads are executed correctly and consistently across the entire network. This involves handling aspects such as task scheduling, load balancing, resource allocation, fault tolerance, and monitoring. Distributed orchestration enables seamless scaling of a nnApp’s specialized network and helps maintain the overall performance, reliability, and consistency of the Blockless platform.

In the Blockless network, fault tolerance and distributed orchestration play vital roles in ensuring the reliable operation of nnApps in a decentralized and verifiable manner. With a set of advanced node selection, task distribution and node ranking algorithm, Blockless provides a robust platform for developers to build and deploy nnApps with the confidence that their applications will remain available and performant, even in the face of unexpected failures or disruptions.

Blockless protocol/network :

Home
Protocol
Networking
Overview
Overview
The Blockless Networking Module (b7s) is the core infrastructure that connects nodes within the Blockless Network and streamlines the distribution of work across the system. It is designed to provide a secure and verifiable execution environment for distributed computing tasks.

The Blockless Networking module (b7s) is open-sourced and can be found on GitHub(opens in a new tab).

Goal
As a purpose-built platform that ensures nnApp fault-tolerance and execution verifiability, the objective of the Blockless Networking Module is to create an efficient execution system that enables the selection of the optimal group of nodes based on nnApp workload requirements, and to execute tasks securely and efficiently within the individual specialized networks of nnApps.

Networking Procedure Overview

The entire networking process, from receiving a task to delivering the result to its intended destination, can be divided into two key steps:

Selection and Distribution: Upon receiving workload from a nnApp, the network processes its workload manifest, which outlines the execution requirements. Based on this information, the network selects a group of nodes that fulfill the specified criteria and distributes the workload with randomly and evenly for execution.

Execution and Consensus: The chosen nodes form a task-specific execution subnetwork, where they carry out the assigned work according to the task setup. A modular consensus mechanism, such as data aggregation, pBFT, Raft, or zero-knowledge proofs, can be integrated into the execution subnetwork to validate the accuracy and integrity of the execution process.

These steps work cohesively to ensure that the Blockless Network can effectively handle various distributed computing tasks while maintaining high levels of security, verifiability, and performance.

Node Selection

In the Blockless Networking (b7s) module, node selection is a crucial step that ensures the efficient and effective operation of nnApps. The selection process aims to identify the optimal network contributors from a pool of devices, taking into account various user-defined filters such as computational capacity, storage availability, and network latency.

Once the selection process is complete, a group of optimal nodes is obtained. This node group serves as the input for the distribution algorithm, which randomly selects the requested number of nodes from the node group for the formation of a nnApp’s network.

In this section, we will focus on the node selection step, exploring the criteria and considerations involved in determining the most suitable nodes for a specific nnApp.

Node Selection Algorithm

The node selection algorithm in the b7s module is built upon two primary components:

Calculating suitability scores: This step quantifies each node's adherence to user-defined filters, providing a numerical assessment of how well the node meets the specified criteria.

Applying a simulated annealing process: This step facilitates a balance between exploration (discovering diverse solutions) and exploitation (refining the best solution found thus far), ensuring a robust and efficient node selection process.

By integrating these two components, the node selection algorithm effectively navigates the search space, identifying the most appropriate node to handle a given task based on the specified filters. This approach ensures that the b7s module maintains high performance, responsiveness, and adaptability, enabling users to leverage the full potential of the distributed network.

Suitability Score Calculation

The suitability score is a crucial component in the node selection algorithm, as it provides a quantitative measure of how well each computer in the pool aligns with the user-defined filters. This numerical assessment enables the algorithm to compare and rank computers based on their adherence to the specified criteria.

To calculate the suitability score (

S
S) for each computer, the following formula is employed:

S
i
=
∑
f
k
(
v
(
i
,
k
)
)
S
i

 =∑f
k

 (v
(i,k)

)
In this formula:

i
i represents the index of a computer in the pool.
k
k denotes the index of a filter applied by the user (e.g., computational power, memory, or network latency).
v
(
i
,
k
)
v
(i,k)

  is the value of the k-th filter for the i-th computer, indicating how well the computer meets that particular filter's requirements.
The function
f
k
f
k

  maps the k-th filter value to a score, which can be either 0 or 1, depending on the filter value type:

Boolean filters: If the filter value is a boolean (e.g., whether a computer has a specific feature), the score is calculated as follows:
f
k
(
v
(
i
,
k

)
)
=
1
f
k

 (v
(i,k)

)=1 if
v
(
i
,
k
)
v
(i,k)

  matches the filter value (i.e., the computer has the desired feature); otherwise, the score is 0.

Range filters: If the filter value is an object with "min" and "max" properties (e.g., a specified range for computational power), the score is calculated as follows:
f
k
(
v
(
i
,
k
)
)
=
1
f
k

 (v
(i,k)

)=1 if
v
(
i
,
k
)
v
(i,k)

  falls within the specified range (i.e., the computer meets the minimum and maximum requirements); otherwise, the score is 0.

The suitability score calculation enables the node selection algorithm to assign a numerical value to each device based on how closely it adheres to the user's (nnApp’s) requirements. This quantitative assessment forms the foundation for the subsequent simulated annealing process, which searches for the most suitable device by balancing exploration and exploitation.

Simulated Annealing Process

The simulated annealing process serves as an essential aspect of the node selection algorithm, facilitating the search for the most appropriate device in the pool while maintaining a balance between exploration and exploitation. This technique is grounded in two main formulas:

$$\Delta E = S(i, candidate) - S(i, current)$$

ΔE=S(i,candidate)−S(i,current)

$$P_{accept} = min(1, exp(-\Delta E /$$

T
)
)
P
accept

 =min(1,exp(−ΔE/T))
In these two formulas:

S
(
i
,
c
a
n
d
i
d
a
t
e
)
S
(i,candidate)

  denotes the suitability score of a randomly chosen candidate device.
S
(
i
,
c
u
r
r
e
n
t
)
S
(i,current)

  represents the suitability score of the currently selected device.
Δ
E
ΔE signifies the difference in suitability scores between the candidate and the current device.
T
T represents the temperature parameter, which begins at 10 and is reduced by a factor of 0.99 per iteration.
P
a
c
c
e
p
t
P
accept

  is the probability of accepting the candidate device as the new current device.
During the simulated annealing process, the algorithm iteratively selects a random candidate device and calculates its acceptance probability based on the

difference in suitability scores and the current temperature. If a random number between 0 and 1 is less than the acceptance probability, the current device is replaced by the candidate device.

As the temperature decreases with each iteration, the algorithm becomes more inclined to accept candidate devices with higher suitability scores, eventually converging towards the most suitable device in the pool. This method ensures that the node selection algorithm can explore various devices before settling on the optimal choice according to the specified filters. The simulated annealing process effectively balances the exploration and exploitation aspects of the selection process, resulting in an efficient and robust device selection algorithm.

Node Selection Process
Given the node selection algorithm, the network operates through the following stages to identify the most suitable device based on user-defined filters:

Filtering: Apply user-provided filters, such as computational power, storage, or network latency, to create a list of eligible devices that meet the specific requirements.

Initial temperature: Set the initial temperature (control) parameter and begin the loop. The temperature plays a crucial role in controlling the balance between exploration and exploitation in the simulated annealing process.

Candidate selection: Within the loop, randomly choose a candidate device and compare its suitability score with the current device's score. This comparison helps in evaluating the devices' adherence to the user-defined filters.

Acceptance probability calculation: Compute the acceptance probability for the candidate device, taking into account the temperature. The acceptance probability determines whether the candidate device is likely to replace the current device.

Candidate acceptance: If the acceptance criteria are met, meaning the candidate device has a higher probability of being suitable based on the user-provided filters, replace the current device with the candidate device.

Loop repetition: Continue the loop, gradually decreasing the temperature until it reaches a minimum value. As the temperature decreases, the algorithm becomes more selective, focusing on devices with higher suitability scores.

Best device selection: Upon reaching the minimum temperature, select the best device based on the user-defined filters, ensuring an optimal choice for the task at hand.

This algorithm effectively balances exploration (discovering a variety of potential solutions) and exploitation (refining the best solution identified thus far) to create an efficient and robust device selection process tailored to the user's specific requirements.

Deployment and Execution
Upon the conclusion of the selection and distribution phases, the task is ready for deployment and execution.

Blockless utilizes a two-stage deployment and execution process, allowing the execution binary to be efficiently installed and run on a set of randomly selected devices for a nnApp. This process first involves the notification of

deployments, which initiates a pre-cache procedure. Subsequently, a runtime environment is instantiated for task execution.

Acquiring the Execution Binary (Pre-Cache)
During the selection phase, a roll call is initiated before the suitability score calculation and simulated annealing process take place. The task message (
D
m
s
g
D
msg

 ), along with the task manifest (
A
A) containing the bundle address (IPFS CID) and content (
C
C), is broadcasted across the network using the GossipSub protocol.

Prior to adding the nodes to the selection and distribution process, or the roll call queue, the presence of the execution binary cache on the node's local machine is checked.

If the cache is not available, the node retrieves the execution binary from either publisher (IPFS by default) based on the task message or from the nearby nodes. Once completed, the node can participate in the roll call, as well as the ensuing selection and distribution processes.

Launching the Execution Binary
Once a node is selected based on the task criteria and receives execution rights from the distribution phase, a dedicated runtime environment is established for this specific task. After the execution is finished, the runtime environment is promptly terminated.

For a comprehensive understanding of the Blockless Runtime Environment, please refer to the Runtime Environment section.

Dynamic Consensus
Blockless introduces a dynamic and pluggable consensus mechanism, offering a modular approach to consensus formation in P2P networks.

Simply put, nnApp developers can choose from various consensus schemes, such as data aggregation, pBFT, Raft, or ZK, and apply them to their dedicated, specialized network built with Blockless. This approach provides optimal security and flexibility for each and every particular workload a nnApp may require.

This dynamic consensus design enables developers to adapt different consensus models based on the task, network conditions, requirements, and preferences. As a result, the network maintains efficiency, security, and scalability while accommodating diverse use cases and applications.

Built on top of the libp2p(opens in a new tab) stack, the dynamic consensus mechanism benefits from a robust and modular foundation for P2P communication. Nodes use libp2p to establish direct connections with other nodes in the network, which facilitates efficient and secure data exchange.

Leveraging this innovative consensus mechanism, nodes can engage in dynamic consensus formation, following rules and protocols specified by the chosen

consensus model. This process unfolds in several steps:

The nnApp developer outlines the preferred consensus model for a particular task within the configuration file.
Based on these specifications, nodes (based on the consensus mechanism, nodes can be execution nodes or stand-alone consensus nodes) create direct connections and ready themselves for consensus formation. Each node executes the consensus within a newly instantiated runtime environment, ensuring secure and efficient processing.
Following several rounds of interaction, the nodes reach a consensus result. Upon reaching a consensus, nodes return the results according to the flow dictated by the consensus model. This may involve sharing results with other nodes, updating local data structures, providing consensus proof to a data availability (DA) layer, or initiating additional processes and actions.

Advantages of the Dynamic Consensus Design
A dynamic consensus mechanism offers several benefits for nnApps:

Flexibility: By enabling tasks to dynamically adopt various consensus models, the mechanism ensures optimal performance and functionality based on specific needs and network conditions of the nnApp.
Scalability: The capacity for nodes to adapt consensus models dynamically contributes to the nnApp's growth and resilience, promoting overall scalability.
Customizability: Thanks to its modular design, the pluggable consensus mechanism empowers nnApp developers to create and implement bespoke consensus models that cater to distinct requirements and use cases.
Implementation of the Dynamic Consensus
Blockless takes a randomized approach for implementing dynamic consensus mechanisms for nnApps. This implementation approach offers several advantages for task execution and resilience within the Blockless Network:

Targeted Node Selection: The process broadcasts a roll call message with specific attribute requirements, ensuring that only nodes possessing the necessary capabilities and resources participate in consensus formation. This optimizes network performance, particularly for resource-intensive consensus/validation protocols like zero knowledge.
Dynamic Cluster Formation: The selection and roll call process allows for the dynamic creation of clusters based on execution factors, such as the number of nodes, failover and resiliency types, and consensus factors. This adaptability caters to varying task requirements and network conditions.
Resilience and Failover: By accounting for failover and resiliency types, the randomized selection-based process enables the consensus subnetwork to maintain operational efficiency and recover from node failures or disruptions.
Customizable Consensus Topology: During task execution, nodes adhere to the requested consensus topology, providing flexibility and adaptability for different use cases and network requirements.

Home
Protocol
Networking
Communication
Node Communication
Having outlined the overall process of the networking procedures involved in a task's life cycle, such as selection and distribution, we now introduce the fundamental component that enables all of the mechanisms described in the networking section.

Blockless Network is built upon the GossipSub Peer-to-Peer (P2P) communication protocol, which combines both gossip protocol and mesh-based routing.

This protocol involves exchanging messages between nodes within a mesh sub-network and periodically gossiping with nodes outside the mesh sub-network. This approach ensures efficient and reliable message dissemination among nodes.

Message Propagation Model
In the Blockless Network, message propagation can be represented using a mathematical model that takes into account various factors affecting the speed and efficiency of information exchange. The equation for this model is as follows:

$$PropagationTime = BaseDelay + RTT * \log(N) / \log(D)$$

where:

Propagation

T
i
m
e
PropagationTime represents the duration required for a message to propagate through the network.
B
a
s
e
D
e
l
a
y
BaseDelay accounts for the inherent delay in the GossipSub protocol, encompassing message encoding, decoding, and processing.
R
T
T
RTT (Round-Trip Time) denotes the average duration for a message to travel from one node to another and back, reflecting the network's latency.
N
N stands for the total number of nodes present in the network.
D
D refers to the average node degree, which is the average number of neighbors a node has in the network.

Node Communication Implementation

Blockless' general communication implementation is comprised of several steps, each contributing to the overall efficiency, integrity, and performance of the network:

Peer discovery: As a new node joins the network, it needs to discover other nodes to establish communication channels. This discovery process relies on the Distributed Hash Table (DHT) and additional discovery mechanisms. After identifying a set of root peers, the node proceeds to establish connections with them.

Message publishing: Nodes within the network can publish messages, initially sending them to their connected peers. The recipient peers then forward the message to a subset of their own connected peers, enabling the message to propagate throughout the network until it reaches all participating nodes.

Message validation: In order to preserve the network's integrity, nodes validate incoming messages before forwarding them. This validation process examines the message format and verifies the sender's authorization to publish the message. This effectively prevents spam and malicious content from spreading throughout the network.

Message aggregation: Blockless employs a message aggregation scheme to optimize network efficiency by reducing redundancy and conserving bandwidth. Nodes maintain a record of previously encountered messages and only forward new messages to their peers.

Peer scoring: Blockless incorporates a peer scoring system to monitor node behavior and penalize misbehaving nodes. Nodes receive scores based on their performance and adherence to network protocols. If a node's score falls below a specified threshold, it may be disconnected from the network to maintain overall performance and security.

Fanout control: To manage network traffic and prevent excessive message propagation, Blockless utilizes a fanout mechanism. Nodes forward messages to a limited number of their connected peers, ensuring efficient message dissemination without overburdening the network.

Home
Protocol
Runtime Environment
Blockless Runtime Environment

The Blockless Runtime Environment is a powerful and versatile system built on WebAssembly (WASM) technology. It offers a secure and isolated sandbox for executing arbitrary tasks using various programming languages, such as C/C++, Rust, Swift, AssemblyScript, and Kotlin. The Blockless Runtime Environment also supports mixing these languages, enabling flexibility and interoperability in software development.

Key Features and Benefits

Optimized Code Generation and Execution

The Blockless Runtime Environment includes a code generator designed to quickly produce high-quality machine code. It is optimized for efficient instantiation, low-overhead transitions between the embedder and WASM, and the scalability of concurrent instances.

Lightweight and Universal Compatibility

The Blockless Runtime Environment is lightweight and capable of running on any device that supports a browser. This feature enables idle personal devices to join and serve the network effortlessly, contributing to the creation of a self-sovereign web.

Customizable Configurations

Users can configure the runtime using a customizable file, which allows for additional restrictions on WebAssembly beyond its basic guarantees. These restrictions can include limitations on CPU and memory consumption, tailoring the runtime to better suit the user's requirements.

WebAssembly System Interface (WASI) Support

The Blockless Runtime utilizes the WebAssembly System Interface (WASI), providing a standardized method for interacting with the host operating system. WASI supports system calls and extensions that allow WebAssembly modules to interact safely and efficiently with the file system, network, and other system or service level resources.

Dynamic Extension System

The WASI interface also features a dynamic extension system, enabling extensions to be added to the interface without modifying the WASM module. This functionality allows the Blockless Network to integrate existing service clients using Common Gateway Interface (CGI) technology.

Security

The Blockless Runtime Environment is designed with security-first principle. For further details, please refer to the security section

Networking
Overview
Node Selection
Randomized Distribution
Deployment and Execution
Dynamic Consensus
Communication
Runtime Environment
Security
Overview
Runtime Security
Consensus And Result Verifiability

Secret Management
Extension
Account
Nodes
Home
Protocol
Security
Overview
Blockless Security Parameter
Overall deployment security in the Blockless network can be viewed from 3 perspectives:

WASM Runtime, or where your code is executed
Verifiability methods, or how your code is executed
Secret management, or how private information (passwords, etc.) is handled

Home
Protocol
Security
Runtime Security
Blockless Runtime Security
Blockless Runtime is a standalone runtime for WebAssembly (WASM), which is a binary instruction format designed as a portable target for the safe and efficient execution of untrusted code.

Blockless Runtime ensures the security of the sandboxed environment using several key principles and mechanisms:

WebAssembly design: WebAssembly itself is designed with security in mind. Its low-level virtual machine uses a stack-based architecture and has no direct access to the host's memory or system resources. Wasm modules can only access memory and resources provided explicitly by the host, which limits the potential attack surface.
Linear memory and bounds checking: WebAssembly uses a single, contiguous block of memory known as linear memory. This memory is isolated from the host system's memory, and all accesses to it are checked to ensure they are within bounds. This prevents buffer overflows and other memory-related vulnerabilities.
Validation: Blockless Runtime validates WebAssembly modules before execution, ensuring that they conform to the WebAssembly specification. This process checks for issues like malformed binaries, invalid opcodes, or type mismatches, helping to prevent potential security risks.
Capability-based security model: Blockless Runtime follows the principle of least privilege by providing only the necessary capabilities to a wasm module. This means that a wasm module can only access resources (like memory, functions, or system calls) that have been explicitly granted to it by the host. This helps to prevent unauthorized access to system resources or sensitive data.
Sandboxing: Blockless Runtime isolates the execution of wasm modules within a secure sandbox, which limits the impact of potential vulnerabilities. Even if an attacker manages to exploit a wasm module, they would still be confined within the sandbox and unable to compromise the host system.
JIT (Just-In-Time) Compilation and CFG (Control Flow Guard): Blockless Runtime uses a JIT compiler to transform WebAssembly bytecode into native machine code. This allows it to implement security measures such as Control Flow Guard, which helps protect against code reuse attacks by ensuring that indirect function calls only target valid locations.
Continuous updates and community involvement: Blockless Runtime is an open-source project with a dedicated community working together to identify and resolve security issues. Regular updates ensure that any vulnerabilities discovered are quickly patched, keeping the runtime secure.

Home
Protocol

Security
Consensus And Result Verifiability
Consensus and Result Verifiability
In an open and trustless system like Blockless, ensuring the correctness and verifiability of execution and data is paramount. To achieve this, Blockless verifies the execution of WASM code through a combination of consensus and zk-SNARK proofs.

Consensus
However, different systems and applications may require varying levels of security and performance. To accommodate these diverse requirements in the production setting, Blockless offers several system-level consensus modules, including data aggregation (off-chain reporting), pBFT, and Raft. Developers can choose from these options based on the desired security and performance trade-offs.

For example, a financial application handling millions of dollars would prioritize security and assurance for its underlying computations and executions, often sacrificing performance. In contrast, an automation system updating smart contracts based on real-world events may require extremely fast execution, opting for the results from the quickest responding worker. In this context, a security mechanism involving global consensus among hundreds of nodes would not suffice.

Result Verifiability
In addition to consensus, developers can choose to generate zk-SNARK proofs, which provide mathematical evidence for the validity and correctness of their execution and results. Blockless currently leverages zkWASM technology to generate zk-SNARK proofs directly from users' WASM binaries, uploading the generated zero-knowledge proofs to designated verifiers on-chain. This added layer of verifiability ensures that the integrity of the computations remains uncompromised, providing users with confidence in the accuracy and trustworthiness of the executed tasks.

Home
Protocol
Security
Secret Management
Secret Management
Blockless employs a robust secret management system to securely store and retrieve sensitive data required for nnApps. This system is built on a combination of threshold signature Multi-Party Computation (MPC) networks and decentralized storage technologies, ensuring both confidentiality and integrity.

Multi-Party Computation Network
A threshold signatures Multi-Party Computation (MPC) Network is a decentralized network of microcontrollers that communicate with each other and store data securely in a tamper-proof manner. The MPC Network facilitates the secure storage and retrieval of sensitive information, ensuring that only authorized parties can access this data during WASM archive execution. This network leverages the power of threshold signatures to protect the sensitive data, requiring a minimum number of participants to cooperate and produce a valid signature or decrypt the data. This approach further enhances the security and resilience of the system against attacks and unauthorized access.

Decentralized Storage
Blockless incorporates decentralized storage solutions like the InterPlanetary File System (IPFS) to securely store secrets while preserving their integrity. When users upload confidential information—such as private keys, API keys, or other sensitive data necessary for WASM archive execution—the data is encrypted and uploaded to the designated decentralized storage location. This approach ensures secure storage of sensitive information, providing enhanced protection

against data breaches, tampering, and unauthorized access.

Secret Retrieval and Execution
When the user’s WASM binary is executed, it requests the necessary secrets from the MPC network. The MPC network retrieves the encrypted secrets from the decentralized storage and decrypts them using the appropriate encryption key. The decrypted secrets are then provided to the WASM binary for execution.

Security and Tamper-Proofing
Blockless's secret management system ensures that sensitive information is stored securely and can only be accessed by authorized parties. The decentralized nature of the MPC Network and decentralized storage guarantees a tamper-proof system resistant to attacks, providing users with confidence in the secure handling of their sensitive data.

Home
Protocol
Extension
Protocol Extendability
Blockless Extensions enhance the protocol by adding functionalities and services beyond the native capabilities provided by the runtime.

These extensions include accessing Ethereum or Redis, integrating with other programming languages or libraries, and more.

In Blockless, extensions are considered safe by default, allowing node operators to download and execute them securely. Additionally, dynamic querying and enumeration of libraries streamline management and maintain consistent performance.

Dynamic Extension Query
Blockless enables the runtime to query and enumerate available extensions on the local host, enabling dynamic method execution and feature querying. This functionality expands the capabilities of the nnApps built on Blockless, delivering more efficient and scalable solutions.

Example: Redis Extension
The Redis extension improves performance and scalability for nnApps built on the platform. It allows for more efficient storage and retrieval of data. Benefits of using the Redis extension include:

Faster Performance: Redis is an in-memory data store, providing faster read and write speeds compared to traditional disk-based databases.
Scalability: Redis is designed for high scalability, making it suitable for applications handling large amounts of data.
Improved Fault Tolerance: Redis supports replication and data sharding, ensuring data availability even during hardware failures or disruptions.
Lower Storage Costs: As an in-memory data store, Redis reduces overall storage costs associated with large-scale networks.
Improved Data Analysis: Redis offers powerful data analysis and aggregation features, helping developers gain deeper insights into network performance and usage patterns.
The Redis extension can help nnApp builders achieve higher levels of performance, scalability, and reliability in using Blockless, especially for large-scale, mission-critical use cases.

Home
Protocol
Account
Account System
The Blockless account system offers a secure and user-friendly solution for managing funds and conducting transactions within the network. It employs an on-

chain custodial account model using smart contracts to handle user deposits, payments, and other actions.

Key Features
Web3 Native
Each Blockless account is associated with a unique Web3 address.

On-Chain, Automated Fund Management and Payment
Payment process for users' calls and invocations are automated via on-chain custodial smart contract, with fees being deducted directly from their account balance.

Account Setup
To set up a Blockless account, users generate an authorized on-chain custodial account associated with their Web3 address. This Web3 address serves as a unique identifier for the account and is used for various purposes, including service payment, request authorization, and account access.

Upon depositing funds into the smart contract, users can leverage Blockless to create specialized networks for their network neutral applications, with the associated fees being automatically deducted from their account balance. By utilizing a smart contract for managing funds, the Blockless account system ensures transparency, security, and ease of use.

Testnet Notice
Blockless currently supports all EVM and Cosmos blockchain addresses. Users can connect to Blockless with MetaMask, Keplr Wallet, and Martian Wallet. Please note that the payment system is currently disabled during testnet.

Home
Protocol
Nodes
Ranking
Node Ranking
A crucial factor in determining a node's task selection probability within the Blockless network is its trust score, or node ranking. The trust score serves as an indicator of a node's trustworthiness and reliability compared to other nodes in the network.

To devise a node's trust score, Blockless employs the Elo rating system, developed by Hungarian-born American physicist Arpad Elo. Originally designed for ranking players in two-player games like chess, the Elo rating system is a skill-based system that measures the likelihood of a player winning a game against another player. In the context of Blockless, "winning a game" refers to the successful completion of a computational task.

The Elo rating system is adapted to assess and rank nodes based on their attributes and the probability outcome of completing computational tasks. This adaptation enables the system to account for factors such as processing power, memory capacity, energy efficiency, and overall performance in benchmark tests. By employing the Elo rating system, Blockless can dynamically and objectively rank nodes, ensuring that tasks are allocated to the most trustworthy and reliable nodes within the network. This, in turn, helps optimize the efficiency and security of the Blockless system.

Advantages of Applying Elo's System to Blockless Network
The Elo rating system offers several benefits when applied to the Blockless Network, making it a robust and flexible solution for ranking nodes based on their performance:

Dynamic ranking: The Elo rating system is inherently dynamic, making it well-suited for ranking nodes in the Blockless Network, where technology and performance are constantly evolving. The system can easily adapt to the

introduction of new nodes and changing performance levels, ensuring the ranking remains up-to-date and relevant.
Objective comparison: The pairwise comparison approach enables a more objective comparison between nodes, as it directly measures their performance in head-to-head contests. This helps reduce the influence of subjective factors and biases on the ranking process, promoting a fair and accurate assessment of node performance.
Customizable criteria: The Elo rating system can be tailored to specific needs by incorporating additional attributes and outcomes of computing or by applying different weightings to these factors. This adaptability allows for a customized ranking system that addresses the unique requirements of various users, industries, or use cases.
Versatility: The Elo rating system can be applied to rank nodes across various segments and categories, from personal devices to high-performance computing clusters. This versatility makes it a valuable tool for evaluating and comparing computing performance across a wide range of applications.
Adapting Elo's System for Computer Ranking in Blockless
To effectively apply the Elo rating system for ranking nodes in the Blockless Network, it is essential to understand several key components of the system in the context of node performance:

Defining attributes and outcomes of computing: The first step involves identifying relevant attributes and outcomes of computing that can be used to compare and rank nodes. These factors include processing power, memory capacity, energy efficiency, overall performance in benchmark tests, and the ability to complete tasks in a timely manner.
Pairwise comparison: Drawing from the original Elo rating system, the adapted method requires pairwise comparison of nodes. To achieve this, a set of standardized tasks or benchmarks is defined, enabling the comparison of two nodes' performance in a head-to-head contest. The performance of each node in completing these tasks influences its rating.
Performance expectation and updating ratings: In line with the principles of the Elo rating system, the performance expectation of each node is calculated based on their current ratings. After the head-to-head comparison, the ratings are updated according to the actual outcome of the comparison, taking into account the performance expectation.
The Blockless node ranking process is iterative, as nodes are continuously compared in various combinations to refine their ratings. This ongoing adaptation ensures that the ranking remains up-to-date as new nodes join the network or existing nodes improve their performance. This process optimizes task allocation and enhances the overall efficiency and reliability of the Blockless Network.

Adapting Elo's System with the Total Points Algorithm
The traditional Elo's system is defined as follows:

$$R_{NewA} = R_{CurrentA} + K \times$$

(
A
c
t
u
a
l
S
c
o
r
e
A
−
(
1
/
(
1
+
1
0
(
(
R
C
u
r
r
e
n
t
B
−
R
C
u
r
r
e
n
t
A
)
/
400
)
)
)
)
RNew
A

 =RCurrent
A

 +K×(ActualScore
A

 −(1/(1+10
((RCurrent
B

 −RCurrent

A

)/400)
)))

where

R
N
e
w
A
RNew
A

: The updated rating of node A after the contest.
R
C
u
r
r
e
n
t
A
RCurrent
A

: The current rating of node A before the contest.
R
C
u
r
r
e
n
t
B
RCurrent
B

: The current rating of node B before the contest.
A
c
t
u
a
l
S
c
o
r
e
A
ActualScore
A

: The actual score of node A in the contest (usually 1 for a win, 0.5 for a draw, and 0 for a loss).
K
K: A constant value, usually set between 10 and 60, that determines how much the rating should change after a contest. Higher values of K will result in larger rating changes.
(

1
/
(
1
+
1
0
(
(
R
C
u
r
r
e
n
t
B
−
R
C
u
r
r
e
n
t
A
)
/
400
)
)
)
(1/(1+10
(
 (RCurrent
B

 −RCurrent
A

)/400))): Computes the expected score of node A based on the current ratings of nodes A and B. If the actual score of node A is higher than the expected score, node A's rating will increase, while if it's lower, node A's rating will decrease. The magnitude of the change depends on the value of K and the difference between the actual and expected scores.
The given algorithm is a generic scoring system used to compute the total points based on various factors. In this section, we will adapt the algorithm to evaluate computing performance considering the completeness of computing, time of response, and machine attributes matching compute requests. We will also discuss the algorithm's applicability in the context of computing performance assessment.

The adapted algorithm for evaluating computing performance can be represented as follows:

T
o
t
a
l
P
o

```
i
n
t
s
=
0
TotalPoints=0
T
o
t
a
l
P
o
i
n
t
s
=
T
o
t
a
l
P
o
i
n
t
s
+
P
o
s
i
t
i
o
n
P
o
i
n
t
s
(
P
)
TotalPoints=TotalPoints+PositionPoints(P)
T
o
t
a
l
P
o
i
n
t
s
=
T
o
t
```

alPoints+(S*StageWinPoints)
TotalPoints=TotalPoints+(S∗StageWinPoints)
TotalPoints=TotalPoints+LapsLedPoint

s
(
L
)
TotalPoints=TotalPoints+LapsLedPoints(L)
T
o
t
a
l
P
o
i
n
t
s
=
T
o
t
a
l
P
o
i
n
t
s
+
P
l
a
y
o
f
f
P
o
i
n
t
s
TotalPoints=TotalPoints+PlayoffPoints

where:

C
o
m
p
l
e
t
e
n
e
s
s
P
o
i
n
t
s

(
C
)
CompletenessPoints(C) represents the points awarded for the completeness of computing tasks.
R
e
s
p
o
n
s
e
T
i
m
e
P
o
i
n
t
s
(
T
)
ResponseTimePoints(T) represents the points awarded based on the time of response.
A
t
t
r
i
b
u
t
e
M
a
t
c
h
P
o
i
n
t
s
(
A
)
AttributeMatchPoints(A) represents the points awarded for the machine attributes that match compute requests.
B
o
n
u
s
P
o
i
n
t
s

BonusPoints represents additional points based on other relevant factors.
Algorithm Application to Computing Performance Assessment:
The adapted algorithm allows for the evaluation of computing performance using a comprehensive scoring system. The various components of the algorithm are explained:

Completeness of Computing
(
C
)
(C): This factor measures the extent to which a computer can perform a given set of tasks. The algorithm assigns points based on the percentage of tasks completed successfully. A higher percentage of task completion results in more points, indicating better performance.
Time of Response
(
T
)
(T): The time of response is a crucial factor in assessing the performance of a computer. The algorithm assigns points inversely proportional to the time taken to complete tasks, with faster response times receiving higher points. This rewards computers that can deliver results more quickly.
Machine Attributes Matching Compute Requests
(
A
)
(A): In addition to the completeness of computing and response time, the algorithm also considers the relevance of the computer's attributes to the specific compute requests. Points are awarded for each attribute that matches the compute request requirements, with a higher total indicating a better-suited computer for the given tasks.
Bonus Points: The algorithm may include additional bonus points based on other relevant factors that influence computing performance. These factors can be tailored to specific use cases or industries, allowing the algorithm to be customized as needed.
Execution Node Ranking Flow Overview
This diagram shows how the Elo ranking of Execution Nodes is updated based on their results.

Untitled

The process of updating the Elo ranking involves several steps that can be objectively described.

Firstly, the tasks are grouped into a map, with each task associated with its respective results. This enables the tracking of the different types of tasks performed and their outcomes.

Secondly, the results are reduced into another map, with each result linked to the computer that executed it. This step determines which computer performed each task and how well they performed it.

To calculate the Elo ranking for each computer, the results for each computer are retrieved, and the average performance score is calculated.

If a computer has not performed any tasks, its Elo ranking is not decreased. This is because the lack of tasks performed does not necessarily indicate a weaker performer but a lack of opportunity to perform tasks.

Once the average performance score for each computer is calculated, it is used to update the Elo ranking. The update involves a mathematical formula that considers the expected performance of a computer based on the Elo rankings of the other computers and compares it to the actual performance. The difference between the expected performance and the actual performance is used to update

the Elo ranking for each computer.

Finally, the updated Elo ranking for each computer is returned, allowing for the determination of the better-performing computer and by how much.

Home
Network
Overview
Blockless Network
Blockless Network implements the Blockless Protocol and is a platform designed to launch, integrate, and secure specialized trustless networks. These networks, in turn, support the operation of Network Neutral Applications (nnApps).

nnApp Architecture
At the heart of each specialized trustless network, and by extension, each Network Neutral Application (nnApp) is what we term as a "function". Within the framework of the Blockless Network, every function encompasses three foundational elements:

Code: This refers to the specific set of instructions or script that will be executed.
Node Group: This is a collection of nodes responsible for executing the code. By default, every function must specify its own node group configuration, with the most fundamental configuration detail being the number of nodes.
Consensus: This is the mechanism or protocol that verifies and validates the execution of the function.
In a fully-fledged nnApp design, multiple functions collaborate, interweaving their operations to fulfill the application's overarching objectives. This collaborative framework implies that a single nnApp could be serviced by several node groups, potentially comprising distinct nodes. Furthermore, different consensus mechanisms may be deployed in tandem to ensure the robustness and security of a single nnApp

Home
Network
Quick Start
How to build with Blockless
This guide will walk you through logging in to the Blockless Network to deploying your first application.

Try the Blockless Dashboard
The quickest way to experiment with Blockless Functions is through the Blockless Dashboard(opens in a new tab). The only thing you need is to link your Web3 wallet and GitHub account. After that, you can deploy a sample project to preview and test a Functions script immediately.

Start a New Project with the Blockless CLI
The Blockless command-line interface (CLI), allows you to create, test, and deploy your Functions projects.

Install the Blockless CLI
To install the Blockless CLI, you can either use curl or wget.

Installing Blockless CLI with curl:

```
sudo sh -c "curl
<https://raw.githubusercontent.com/BlocklessNetwork/cli/main/download.sh> |
bash"
```

Installing Blockless CLI with wget:

```
sudo sh -c "wget
```

<https://raw.githubusercontent.com/BlocklessNetwork/cli/main/download.sh> -v -O download.sh; chmod +x download.sh; ./download.sh; rm -rf download.sh"

To install on Windows, go to the releases page(opens in a new tab) on GitHub and download the x86 version of the Blockless CLI.

Get Familiar with Blockless CLI Commands
To see a list of available commands, you can run the bls or bls help command:

```
bls help
```

Log In to Your Account
To log in to the Blockless Network, run:

```
bls login
```

A pop up window will take you to connect your Web3 wallet.

Create a New Functions Project
To create a new Functions project, run:

```
bls functions init
```

In your terminal, you will be asked a series of questions related to your project.

Test Your Project Locally
In your project directory, the bls function init command has generated the following files:

bls.toml: Your Blockless configuration file.
readme.md: Project readme file
index.ts: A minimal Hello World Function written in TypeScript.
package.json: A minimal Node dependencies configuration file.
asconfig.json: AssemblyScript configuration.
After creating your first Blockless Function, run the bls components install command to install the local runtime environment. This will enable you to test your function locally.

Next, execute the following command to build your function. Once the build sequence is complete, a build folder will appear in your project directory.

```
bls function build
```

To test your function, simply run the bls function invoke command. You will see your function result returned to the terminal after execution.

Deploy Your Project
With your project configured, you can now deploy your Function to the Blockless Network.

```
bls function deploy
```

You will be able to view your deployed function either via the bls function list command or via the Blockless Dashboard.

Next Steps
To do more with Blockless Functions, explore the tutorials.

Blockless utilizes a configuration file called bls.toml to enable customization of the development and publishing setup for your Blockless Function.

Here, you can find a sample bls.toml file that is generated when you run the bls function init command to create a new Hello World project.

```
name = “my-function"
type = "function"
version = "1.0.0"
content_type = "json"

[deployment]
nodes = 4
permissions = [
  "https://bsc-dataseed.binance.org",
  "https://api.coingecko.com",
  "https://redis-domain:port"
]

[build]
dir = "build"
entry = “my-function_debug.wasm"
command = "npm run build:debug"

[build_release]
dir = "build"
entry = “my-function.wasm"
command = "npm run build:release"
```

Top-Level Configuration
In the top-level configuration, you can define your function’s name and version.

```
name = “my-function"
type = "function"
version = "1.0.0"
content_type = "json"
```

name string Required
The name property is a string that represents the unique identifier for your function.

version string Optional
The version property is an optional string that represents the version of your function.

type string Optional
The type property is an optional string that determines the type of deployment. Currently, the type field needs to be "function".

content_type string Optional
The content_type property is an optional string that defines the content type for the response returned by your function. Some type examples are: "json", "html", or "text". If not provided, it defaults to "text".

Deployment Configuration [deployment]
The deployment section is used to specify the deployment settings for the application.

```
[deployment]
nodes = 4
permissions = [
  "https://bsc-dataseed.binance.org",
  "https://api.coingecko.com",
  "https://redis-domain:port"
```

]

nodes integer Optional
The nodes property is an optional integer that represents the number of nodes to be deployed for your function. If not provided, it defaults to 1.

permissions array of strings Conditional
The permissions property is an array of strings that represents the list of extensions (and subsequent URL) your function is allowed to make requests to. This property is required only if your function makes external requests.

Build Configuration [build]
The build configuration is used to specify the build settings for the application.

```
[build]
dir = "build"
entry = “my-function_debug.wasm"
command = "npm run build:debug"

[build_release]
dir = "build"
entry = “my-function.wasm"
command = "npm run build:release"
```

[build] Optional
The optional build [build] section contains the settings for building the function in —-debug mode.

[build_release] required
The build_release [build_release] section is used to specify the build settings for the release or production version of your function.

dir string required
The dir field specifies the directory where the build files will be stored.

entry string required
The entry field specifies the name of the entry point for your function. The entry point is the main function that is executed when the application is run.

command string required
The command field specifies the command that will be used to build the application. This field is used to specify the build command for the debug and release versions of the application.

Home
Network
Extensions
Extending Your Function Capabilities
This document provides an overview of the different types of extensions available for use in your projects. Extensions can be thought of as tools or additional functionality that you can integrate into your code to enhance its capabilities. These extensions can either be internal, external, or CGI-based. In this documentation, we will discuss each of these extension types, their use cases, and examples.

Internal Extensions
Internal extensions are similar to packages or libraries that you can import into your code. They are compiled together with your WebAssembly (WASM) binary at build time, which means they are part of your final compiled project. This provides the advantage of having all the necessary components and dependencies bundled together, ensuring the proper functioning of your code.

Use Cases

When you need specific functionality that is available in a package or library.
When you want to reduce the number of external dependencies.
When you need to optimize your project for speed and performance.
Example
If you are developing a project in Rust and you need to use a JSON parsing library, you can include the serde_json crate as an internal extension. This will compile the library with your project, allowing you to use its features within your code.

```
use serde_json::{Value, Error};

fn parse_json(json_str: &str) -> Result<Value, Error> {
    serde_json::from_str(json_str)
}
```

External Extensions
External extensions, or on-demand extensions, are extensions that are installed and managed by the host. They can be accessed by your function as needed. These extensions are not compiled with your project but are instead provided by the host machine. This can be helpful in situations where you need access to resources or functionality that is not feasible to include in your binary.

Use Cases
When you need access to resources provided by the host machine.
When you need to use functionality that cannot be compiled into your WASM binary.
When you want to leverage existing system resources or tools.
Example
The GETH extension is an example of an external extension. This extension allows you to access an Ethereum node that is present on the host machine. You can use this extension to interact with the Ethereum blockchain, perform transactions, or query data from smart contracts.

CGI Extensions
Common Gateway Interface (CGI) extensions allow you to access packages or libraries that are written in other programming languages. This is useful when you need to use functions or features that are not available in your primary language or when you want to integrate with existing codebases or libraries. CGI extensions essentially export the functions you need and enable you to use them in your own code.

Use Cases
When you need to use a library or package that is written in another programming language
When you want to integrate with existing codebases or tools
When you need access to features or functions that are not available in your primary language
Example
Use the LIT CGI extension to check the validity of a given JWT token:

```
import 'wasi'

import { Console } from 'as-wasi/assembly'
import { cgi } from '@blockless/sdk'

// Check whether the extension is available
function isExtensionAvailable(alias: string): boolean {
      let extensions = cgi.cgiExtendsList()
      let isMatch = false

      if (extensions && extensions.length > 0) {
            for (let i = 0; i < extensions.length; i++) {
                  const extension = extensions[i]
                  if (alias === extension.alias) {
```

```
                        isMatch = true
                    }
                }
        }

        return isMatch
}

// Run the extension
function testLitProtocol(): void {
        let command = new cgi.CgiCommand('lit-extension', [], [])
        let rs = command.exec()

        if (rs === true) {
                Console.log(`Verify JWT`)
                const verifyJwtResponse = command.callMethod('verifyJWT', [
                        'eyJhbGciOiJCTFMxMi0zODEiLCJ0eXAiOiJKV1QifQ.eyJpc3MiOiJMSVQiLC
JzdWIiOiIweGZmZjE3NWMxNGEyOTllZjcwMjdkYTBkMzQ4ZjQzOGUxNTQ4ODBjY2QiLCJjaGFpbiI6Im
V0aGVyZXVtIiwiaWF0IjoxNjc2NDQ0NTcwLCJleHAiOjE2NzY0ODc3NzAsImJhc2VVcmwiOiJsaXQtdG
VzdC5ibHMuZGV2IiwicGF0aCI6Ii81Ym9uanB4NHE2Nmoxbng2c2ZoZjEiLCJvcmdJZCI6IiIsInJvbG
UiOiIiLCJleHRyYURhdGEiOiIifQ.puWAqp82-1OM-
jiHwl2jFroforAU7A5DY_4u9lXZ9KbuPHFcXQB-ovWN9DWfD7DGBf-KxT5_6f5Ii0cHmWi3TAFv-
KAVkIYPAX-r_6tV6_ot2mle8pU7f43O_I_mjxwi'
                ])
                Console.log(verifyJwtResponse)
        }
}

if (isExtensionAvailable('lit-extension')) {
        testLitProtocol()
} else {
        Console.log('Extension not available.')
}
```

Home
Network
Functions Workflow
Overview
Overview
Functions in the Blockless network can be easily integrated with each other. Whether you are writing a private function for your own stack or a public function for the entire community, there are a few important principles to remember for easier function workflow and composability.

Home
Network
Functions Workflow
Requesting Data from Host
Requesting Information from Host
Since the function runs inside the WASM-based runtime environment, the permission system requires you to request host information from the “Memory Extension”.

If you requested permission in the function manifest and the request is granted, host information can be retrieved by importing memory from Blockless SDK. Here's an example of reading the stdin interface from memory:

```
import 'wasi'
import { Console } from 'as-wasi/assembly'
import { memory } from '@blockless/sdk'

// { "results": "1000" }
// cast stdin input to JSON
```

```
let stdin = new memory.Stdin().read().toJSON()

if (stdin) {
      // read the results object out of STDIN
      let results = stdin.get('results')
      if (results) {
            // convert value and add 1000 to it
            let newValue = Number.parseFloat(results.toString()) + 1000
      }
}
```


Passing Data to Other Functions
If you need to pass your execution results to another function, you should return those results to stdout. This interface currently has a 1Mib limit. If your returned results is larger than the 1024Kib, only the first 1024Kib will be transported over the P2P network. If this causes the message payload to be incomplete, the message may not be understood by the Network and may result in errors in your function workflow.

Here's an example of returning a results object, with the value of 1000:

```
import 'wasi'
import { Console } from 'as-wasi/assembly'
import { json } from '@blockless/sdk'

// create a new jsonEncoderlet jsonEncoder = new json.JSONEncoder();

// pop an object onto the stack
jsonEncoder.pushObject('')

// set a string value for the object
jsonEncoder.setString('results', '1000')

// close the object
jsonEncoder.popObject()

// send the results to stdout
Console.log(jsonEncoder.toString())
```

Home
Network
Functions Workflow
Passing Data to Other Functions
Passing Data to Other Functions
If you need to pass your execution results to another function, you should return those results to stdout. This interface currently has a 1Mib limit. If your returned results is larger than the 1024Kib, only the first 1024Kib will be transported over the P2P network. If this causes the message payload to be incomplete, the message may not be understood by the Network and may result in errors in your function workflow.

Here's an example of returning a results object, with the value of 1000:

```
import 'wasi'
import { Console } from 'as-wasi/assembly'
import { json } from '@blockless/sdk'
```

```
// create a new jsonEncoderlet jsonEncoder = new json.JSONEncoder();

// pop an object onto the stack
jsonEncoder.pushObject('')

// set a string value for the object
jsonEncoder.setString('results', '1000')

// close the object
jsonEncoder.popObject()

// send the results to stdout
Console.log(jsonEncoder.toString())
```

Home
Network
Functions Workflow
Passing Data to Other Functions
Passing Data to Other Functions
If you need to pass your execution results to another function, you should return those results to stdout. This interface currently has a 1Mib limit. If your returned results is larger than the 1024Kib, only the first 1024Kib will be transported over the P2P network. If this causes the message payload to be incomplete, the message may not be understood by the Network and may result in errors in your function workflow.

Here's an example of returning a results object, with the value of 1000:

```
import 'wasi'
import { Console } from 'as-wasi/assembly'
import { json } from '@blockless/sdk'

// create a new jsonEncoderlet jsonEncoder = new json.JSONEncoder();

// pop an object onto the stack
jsonEncoder.pushObject('')

// set a string value for the object
jsonEncoder.setString('results', '1000')

// close the object
jsonEncoder.popObject()

// send the results to stdout
Console.log(jsonEncoder.toString())
```

Home
Network
Functions Workflow
Passing Data to Other Functions
Passing Data to Other Functions
If you need to pass your execution results to another function, you should return those results to stdout. This interface currently has a 1Mib limit. If your returned results is larger than the 1024Kib, only the first 1024Kib will be transported over the P2P network. If this causes the message payload to be incomplete, the message may not be understood by the Network and may result in errors in your function workflow.

Here's an example of returning a results object, with the value of 1000:

```
import 'wasi'
import { Console } from 'as-wasi/assembly'
import { json } from '@blockless/sdk'
```

```
// create a new jsonEncoderlet jsonEncoder = new json.JSONEncoder();

// pop an object onto the stack
jsonEncoder.pushObject('')

// set a string value for the object
jsonEncoder.setString('results', '1000')

// close the object
jsonEncoder.popObject()

// send the results to stdout
Console.log(jsonEncoder.toString())
```

Home
Network
Functions Workflow
Passing Data to Other Functions
Passing Data to Other Functions
If you need to pass your execution results to another function, you should return those results to stdout. This interface currently has a 1Mib limit. If your returned results is larger than the 1024Kib, only the first 1024Kib will be transported over the P2P network. If this causes the message payload to be incomplete, the message may not be understood by the Network and may result in errors in your function workflow.

Here's an example of returning a results object, with the value of 1000:

```
import 'wasi'
import { Console } from 'as-wasi/assembly'
import { json } from '@blockless/sdk'

// create a new jsonEncoderlet jsonEncoder = new json.JSONEncoder();

// pop an object onto the stack
jsonEncoder.pushObject('')

// set a string value for the object
jsonEncoder.setString('results', '1000')

// close the object
jsonEncoder.popObject()

// send the results to stdout
Console.log(jsonEncoder.toString())
```

Home
Network
Tutorials
Serverless Todo App
Serverless To-Do App with Blockless Functions
In this tutorial, we will show you how to build a simple todo application using the Blockless Network and a variety of modules and tools. We will cover how to connect to the network using the Blockless CLI, read data from the standard input and environment variables using the memory module, interact with the InterPlanetary File System (IPFS) using the ipfs module, send HTTP requests and receive responses from servers using the http module, and store data in the cloud using the AWS s3 module.

By following the steps in this tutorial, you will learn how to use these powerful tools to build applications that can interact with the Blockless Network and make use of its unique features. You will be able to use this

knowledge to create your own applications and explore the full capabilities of the Blockless ecosystem. You will learn how to:

Connect to the Blockless Network using the Blockless CLI
Read data from the standard input and environment variables using the memory module
Interact with the InterPlanetary File System (IPFS) using the ipfs module
Send HTTP requests and receive responses from servers using the http module
Use Amazon S3 to store data in the cloud using the awss3 module
Build the Todo Application
By the end of this tutorial, you will have a fully functional todo application that you can deploy and run on the Blockless Network. Let’s get started!

Reading from the Standard Input
You can use the memory.Stdin class to read data from the standard input. This can be useful if you want to pass data to your application when it is run.

```
let stdin = new memory.Stdin().read().toJSON();
if (stdin) {
  let results = stdin.get("results");
  if (results) {
    let newValue = Number.parseFloat(results.toString()) + 1000;
  }
}
```

Reading Environment Variables
You can use the memory.EnvVars class to read environment variables. This can be useful if you want to access configuration data or other global values.

```
let envVars = new memory.EnvVars().read().toJSON();
if (envVars) {
  let environmentValue = envVars.get("ENV_VAR_NAME");
  if (environmentValue) {
    Console.log("Hello " + environmentValue.toString());
  }
}
```

Interacting with IPFS
The ipfs module in Blockless allows you to access and manipulate data stored on the IPFS network.

To use the ipfs module in your Blockless application, you need to import it in your main AssemblyScript file:

```
import { ipfs } from "../assembly";
```

Listing Files and Directories
You can use the ipfs.ipfsFileList function to list the files and directories in a given path. The function takes a path as an argument and returns an array of file names.

Here is an example of how to list the files and directories in the root directory:

```
let files = ipfs.ipfsFileList("/");
if (files != null)
  Console.log(`Files and directories: ${files!.toString()}`);
```

Removing Files and Directories
You can use the ipfs.ipfsFileRemove function to delete a file or directory from IPFS. The function takes a path, a flag for whether to delete recursively, and a flag for whether to force the delete as arguments. It returns a boolean indicating whether the delete was successful.

Here is an example of how to delete the file or directory at the path “/1”:

```
let isDeleted = ipfs.ipfsFileRemove("/1", true, true);
Console.log(`Delete successful: ${isDeleted}`);
```

Creating Directories
You can use the ipfs.ipfsCreateDir function to create a new directory in IPFS. The function takes a path and a flag for whether to create the directory recursively as arguments. It returns a boolean indicating whether the directory was created successfully.

Here is an example of how to create the directory at the path “/1”:

```
let isCreated = ipfs.ipfsCreateDir("/1", true);
Console.log(`Directory creation successful: ${isCreated}`);
```

Writing to Files
You can use the ipfs.ipfsFileWrite function to write data to a file in IPFS. The function takes a FileWriteOptions object and a Uint8Array of data as arguments. It returns a boolean indicating whether the write was successful.

Here is an example of how to write the data [65, 66, 67, 68, 69, 70] (representing the ASCII values for the characters ‘ABCDEF’) to a file named “/2.txt”:

```
let filename = "/2.txt";
let wopts = new FileWriteOptions(filename);
let isWritten = ipfs.ipfsFileWrite(wopts, [65, 66, 67, 68, 69, 70]);
 console.log(`Write successful: ${isWritten}`);
```

Reading from Files
You can use the ipfs.ipfsFileRead function to read data from a file in IPFS. The function takes a path, a starting position, and a Uint8Array to store the data as arguments. It returns the number of bytes read.

Here is an example of how to read the data from the file “/2.txt”:

```
let buf = new Array<u8>(1024);
let numBytesRead = ipfs.ipfsFileRead("/2.txt", 0, buf);
let data = String.UTF8.decodeUnsafe(buf.dataStart, numBytesRead);
 console.log(`Read ${numBytesRead} bytes: ${data}`);
```

Getting File Stats
You can use the ipfs.ipfsFileStat function to get information about a file in IPFS, such as its size and creation time. The function takes a path as an argument and returns a FileStat object.

Here is an example of how to get the file stats for the file “/2.txt”:

```
let fstat = ipfs.ipfsFileStat("/2.txt");
if (fstat != null) {
  Console.log(`File stats for "/2.txt":`);
  Console.log(`Size: ${fstat!.size}`);
  Console.log(`Creation time: ${fstat!.ctime}`);
}
```

That concludes the tutorial section on interacting with IPFS using the ipfs module. You should now be able to list, remove, create, write to, read from, and get information about files and directories on the IPFS network.

Sending HTTP Requests
The http module in Blockless allows you to send HTTP requests and receive responses from servers.

To use the http module in your Blockless application, you need to import it in your main AssemblyScript file:

```
import { http } from "../assembly";
```

To send an HTTP request, you can use the http.HttpOpen function. This function takes a URL and an HttpOptions object as arguments and returns an HttpHandle object.

Here is an example of how to send a GET request to the URL “http://httpbin.org/json(opens in a new tab):

```
let handle = http.HttpOpen(
  "http://httpbin.org/json",
  new http.HttpOptions("GET")
);
```

Receiving HTTP Responses
Once you have sent an HTTP request and received an HttpHandle object, you can use various functions of the HttpHandle class to get information about the response.

Getting the Response Headers You can use the HttpHandle.getHeader function to get the value of a specific response header. The function takes a header name as an argument and returns the value of the header as a string.

Here is an example of how to get the “Content-Type” header from an HttpHandle object:

```
let contentType = handle.getHeader("Content-Type");
```

Getting the Response Body
You can use the HttpHandle.getAllBody function to get the entire response body as a string.

```
let body = handle.getAllBody();
```

Closing the Connection
Once you have finished using an HttpHandle object, you should close the connection to the server by calling the HttpHandle.close

```
handle.close();
```

That concludes the tutorial section on sending HTTP requests and receiving responses using the http module. You should now be able to send various types of HTTP requests, get information about the responses, and properly close the connections

Other Features
The http module also provides other features that you may find useful in your projects.

Setting Request Headers
You can set request headers when sending an HTTP request by passing an object with header name-value pairs as the third argument to the http.HttpOpen function.

```
let data = { message: "Hello, world!" };
let handle = http.HttpOpen(
  "http://httpbin.org/anything",
  new http.HttpOptions("POST"),
  { "Content-Type": "application/json" },
  json.JSON.stringify(data)
);
```

Setting Up Amazon S3

Before you can use the awss3 module in your Blockless application, you need to set up an Amazon Web Services (AWS) account and create a bucket in Amazon S3.

Go to the AWS website and create an account.
Follow the instructions in the AWS documentation to create a bucket in Amazon S3.
Make note of your AWS access key ID and secret access key, which you will need to use the awss3 module.

Importing the Module

To use the awss3 module in your Blockless application, you need to import it in your main AssemblyScript file:

```
import { Bucket, S3Configure } from "../assembly/awss3";
```

Connecting to Amazon S3

To connect to Amazon S3, you need to create a S3Configure object with your AWS access key ID, secret access key, and the endpoint of your bucket.

```
let s3Config = new S3Configure(
  "your-access-key-id",
  "your-secret-access-key",
  "your-bucket-endpoint"
);
```

Then, you can create a Bucket object with the name of your bucket and the S3Configure object.

```
let bucket = new Bucket("your-bucket-name", s3Config);
```

Storing Data in Amazon S3

With a Bucket object, you can use the Bucket.putObject function to store data in Amazon S3. The function takes a path

```
let data = [65, 66, 67, 68, 69, 70]; // ASCII values for 'ABCDEF'
let success = bucket.putObject("/path/to/file.txt", data);
if (success) {
  Console.log("Data stored successfully in Amazon S3.");
} else {
  Console.log("Error storing data in Amazon S3.");
}
```

Retrieving Data from Amazon S3

You can use the Bucket.getObject function to retrieve data from Amazon S3. The function takes a path and returns a Uint8Array with the data.

```
let data = bucket.getObject("/path/to/file.txt");
if (data != null) {
  Console.log(`Data retrieved from Amazon S3: ${data}`);
} else {
  Console.log("Error retrieving data from Amazon S3.");
}
```

Listing Objects in a Bucket

You can use the Bucket.list function to get a list of objects in a bucket. The function takes a prefix as an argument and returns a string with the names of the objects separated by newlines.

```
let objects = bucket.list("/path/to/prefix");
if (objects != null) {
  Console.log(`Objects in bucket with prefix "/path/to/prefix":`);
  Console.log(objects);
```

Deleting Objects in a Bucket
You can use the Bucket.deleteObject function to delete an object in a bucket. The function takes a path and returns a boolean indicating whether the delete was successful.

```
let success = bucket.deleteObject("/path/to/file.txt");
if (success) {
  Console.log("Object deleted successfully from Amazon S3.");
} else {
  Console.log("Error deleting object from Amazon S3.");
}
```

That concludes the tutorial section on using the awss3 module to store data in Amazon S3. You should now be able to connect to your bucket, store data, retrieve data, list objects, and delete objects in your bucket.

Building the Todo Application
Creating a New AssemblyScript Project with the Blockless CLI In this section, we will use the Blockless CLI to create a new project and set up an AssemblyScript environment. We will then install the necessary dependencies for our todo application and start building our code.

Follow these steps to create a new AssemblyScript project with the Blockless CLI and set up your development environment:

Open a terminal and navigate to the directory where you want to create your project.
Run the bls function init command to create a new project with a default AssemblyScript configuration.
$ bls function init

The bls function init command will create a new directory with the following structure:

```
my-project/
├── package.json
├── assembly
│   └── index.ts
└── package-lock.json
```

The package.json file contains the dependencies and scripts for your project. The assembly directory contains your AssemblyScript source code, and the package-lock.json file is used to manage the dependencies of your project.
To add the necessary dependencies for your project, open the package.json file and add the following dependencies:
"dependencies": { "as-wasi": "^0.4.0", "as-aws-s3": "^0.4.0", "as-http": "^0.4.0", "as-ipfs": "^0.4.0", "as-memory": "^0.4.0", "as-json": "^0.4.0"},

Save the package.json file and run the npm install command to install the dependencies.
$ npm install

You are now ready to start building your AssemblyScript code in the assembly/index.ts file. You can use the memory, ipfs, http, and awss3 modules to interact with the Blockless Network and build your todo application.
Get Building!
To build the todo application, you will need to use the memory, ipfs, and http modules. You can start by importing these modules in your main AssemblyScript file:

```
import { memory } from "../assembly";
import { json, ipfs } from "../assembly";
import { http } from "../assembly";
```

Next, you can define a Todo class that represents a single todo item. The class

should have a text field for the todo text and a completed field for the completion status. You can also define methods for reading and writing the todo from IPFS and for sending HTTP requests to update the todo status on the server.

```
class Todo {
  text: string;
  completed: boolean;

  constructor(text: string, completed: boolean) {
    this.text = text;
    this.completed = completed;
  }

  // Reads the todo from IPFS
  read(): Todo | null {
    let data = ipfs.ipfsFileRead("/todo.json");
    if (data == null) {
      return null;
    }
    let jsonString = String.UTF8.decodeUnsafe(data.dataStart, data.length);
    let jsonObject = <json.JSON.Obj>json.JSON.parse(jsonString);
    let kvs = jsonObject.valueOf();
    if (kvs == null) {
      return null;
    }
    let text = kvs.get("text");
    let completed = kvs.get("completed");
    if (text == null || completed == null) {
      return null;
    }
    return new Todo(text, completed);
  }

  // Writes the todo to IPFS
  write(): boolean {
    let jsonObject = json.JSON.obj();
    jsonObject.set("text", this.text);
    jsonObject.set("completed", this.completed);
    let jsonString = json.JSON.stringify(jsonObject);
    let data = String.UTF8.encode(jsonString);
    return ipfs.ipfsFileWrite(new ipfs.FileWriteOptions("/todo.json"), data);
  }

// Sends an HTTP request to update the todo status on the server
  update(): boolean {
    let options = new http.HttpOptions("POST", "/update");
    options.setHeader("Content-Type", "application/json");
    let jsonObject = json.JSON.obj();
    jsonObject.set("text", this.text);
    jsonObject.set("completed", this.completed);
    let jsonString = json.JSON.stringify(jsonObject);
    let data = String.UTF8.encode(jsonString);
    options.setBody(data);
    let handle = http.HttpOpen("http://example.com", options);
    if (handle == null) {
      return false;
    }
    let statusCode = handle.getStatusCode();
    handle.close();
    return statusCode == 200;
  }
}
```

With the Todo class defined, you can now create the main function for your application. In this function, you can create a new Todo object and read it from IPFS. You can then display the todo text and a prompt for the user to enter a new status for the todo (either “completed” or “not completed”).

```
export function main(): void {
  let todo = new Todo("Buy milk", false).read();
  if (todo == null) {
    Console.log("Error reading todo from IPFS.");
    return;
  }
  Console.log(`Current todo: ${todo.text} (${todo.completed ? "completed" : "not
completed"})`);
  Console.log("Enter new status (completed/not completed):");
  let stdin = new memory.Stdin().read().toJSON();
  if (stdin == null) {
    Console.log("Error reading from standard input.");
    return;
  }
  let results = stdin.get("results");
  if (results == null) {
    Console.log("Error reading from standard input.");
    return;
  }
  let status = results.toString();
  if (status == "completed") {
    todo.completed = true;
  } else if (status == "not completed") {
    todo.completed = false;
  } else {
    Console.log("Invalid status entered.");
    return;
  }
  if (!todo.write()) {
    Console.log("Error writing todo to IPFS.");
    return;
  }
  if (!todo.update()) {
    Console.log("Error updating todo on server.");
    return;
  }
  Console.log("Todo updated successfully.");
}
```

With the main function defined, you can now build and run your todo application. You can use the Blockless CLI to compile your AssemblyScript code and deploy it to the Blockless Network.

Build

```
$ bls function build
```

Deploy

```
$ bls function deploy
```

To test your application, you can use the bls function invoke command to run the main function and interact with it via the command line.

```
$ bls function invokeCurrent todo: Buy milk (not completed)Enter new status
(completed/not completed):completedTodo updated successfully.
```

Congratulations, you have now built a simple todo application using the Blockless Network! You can continue to build and improve your application by

using other modules and features available in the Blockless ecosystem.

Home
Network
Economics
Token Economics
Token Economics
The Blockless Network's token economic mechanism is founded on the direct interaction between the nnApp developers, who deploys and runs specialized networks on Blockless, and the node operator or contributor, who supplies hardware resources to the networks.

When developers deploy applications, they deposit funds into a smart contract wallet across various blockchain ecosystems, which manages the emission of funds. As the developer's application receives execution requests, the designated funds are transferred to the node contributor.

From the total fee collected from nnApp developers, 80% is allocated directly to the node operator as compensation for their hardware contribution. The remaining 20% is split into two parts: 10% is burned, effectively reducing the token supply and promoting long-term value; the other 10% is deposited into the protocol treasury, which funds network improvements and ongoing development.

Execution Nodes - 80%
Token Burn - 10%
Protocol Improvement Funds - 10%
Node Incentive Distribution
The Blockless Network's randomized node selection and distribution process requires a balanced and fair incentive distribution mechanism that accommodates varying commitment levels of node operators while maintaining network security.

Recognizing that not all nodes can serve the network perpetually or in a long-running manner, Blockless incorporates a novel tiering system, dividing nodes into two categories: time-commitment (full) nodes and freelance (lite) nodes.

Time-commitment nodes require staking as a timed SLA (denoted in hours) is established between the node operator and the network. In exchange, the network guarantees a variable yield for time-commitment nodes and prioritizes their fee distribution. This arrangement ensures that dedicated node operators receive appropriate compensation for their commitment.

Freelance nodes do not require staking, although it may be enforced to enhance network security. Freelance nodes can join and exit the network at their discretion, and they are compensated after all time-commitment nodes are rewarded. This means that freelance nodes may receive no incentives when node resources are abundant, ensuring that only the most active and dedicated nodes receive a significant share of the rewards.

Home
Run a Node
Overview
Blockless Nodes
The Blockless Network is powered by distributed nodes, which are hardware devices contributing CPU/GPU, RAM, and bandwidth to verifiably execute workloads in return for service rewards, forming a peer-to-peer computing network.

Nodes can join the network by installing the node client, which runs the Blockless Runtime, a secure WASM-based execution environment, and communicates with other nodes via the Blockless Networking Module b7s.

Currently, the Blockless Network is in the pre-testnet phase, with ongoing optimizations for each new release and the documentation subject to further

updates.

Home
Run a Node
Roles of Nodes
Roles of Nodes
There are two types of nodes that power the Blockless Network:

Orchestration Node: These nodes run the orchestration algorithm, which automates the matchmaking between workloads and Worker Nodes, ensuring execution security and efficiency across the network. The number of Orchestration Nodes is limited to 70.
Worker Node: These nodes contribute their available hardware resources to execute workloads, based on the automatic orchestration task distribution from the orchestration algorithm. The number of Worker Nodes is uncapped.
Distinct from traditional Proof-of-Stake blockchain networks, the Blockless Network does not depend on validator nodes for ledger-wide global consensus. Instead, it utilizes modular consensus modules. Only the Worker Nodes involved in executing a specific workload form a task-specific dynamic consensus or perform verification using zero-knowledge proof when a single Worker Node executes, to validate the accuracy and integrity of the execution process.

Home
Run a Node
Node Requirements
Node Requirements
To join the network, a node must meet both staking and hardware requirements.

Staking
The Blockless Network supports multi-asset staking, including re-staked ETH and other assets approved by the foundation under DAO governance. Each node tier requires a fixed stake amount based on its hardware capacity, providing economic security and technical security measures.

The network allows both native staking and delegated proof-of-stake. At testnet launch, a communal staking pool will be evenly distributed to node operators following a whitelisting process.

Please note that staking is not enabled in the current release.

Hardware Specification

Node Type	Minimum			Recommended		
	CPU	RAM	Network	CPU	RAM	Network
Orchestration Node	2 vCPU	8 GB	5 Mbps	4 vCPU	16 GB	20 Mbps
General Worker Node	.5 vCPU	1 GB	2 Mbps	1 vCPU	8 GB	5 Mbps
AVS Worker Node	2 vCPU	8 GB	5 Mbps	2 vCPU	16 GB	10 Mbps

Home
Run a Node
Slashing
Slashing
Orchestration Node
These nodes will be penalized for double-signing, resulting in their permanent removal from the node set and a 10% penalty on their staked amount. There is no slashing for downtime, nor are rewards distributed during periods of downtime.

Worker Node
Worker Nodes do not face slashing on their staking principal. The Blockless Secure Runtime and Fault Tolerance mechanisms are designed to maintain continuous correct operation, even in the event of malicious attacks or

disruptions.

However, slashing does occur on the service rewards if a node’s trust score, as defined in the Node Ranking, falls below the threshold level, which is determined based on historical performance aligned with the Service Level Agreement (SLA). The slashed stakes are confiscated and redistributed to the communal staking reward pool.

Worker Nodes will not be slashed for downtime.

Worker Nodes that consistently fail in dynamic consensus or attach incorrect zero-knowledge proofs will receive low rankings, negatively impacting their trust score. This will result in their deprioritization in the selection process for future tasks.

Home
Run a Node
Incentives
Incentives
Reward emission follows an exponential decay function of time, to incentivize stakers to contribute security and node operators to contribute hardware resources to the network.

Node operators can request immediate withdrawal of rewards, which are distributed upon such requests.

Please note that the incentive module is not enabled in the current release.

Home
Run a Node
Service Terms
Service Terms
Orchestration Nodes have to commit to a 30-day cycle minimum.

Worker Nodes have the flexibility to operate on a freelance basis with no time commitment or can choose to commit to a multiple of 30-day period(s). Upon deciding to exit, a time-based Worker Node must submit a withdrawal request and then wait 30 days to be removed from the active set, after which it can unstake and withdraw all accrued rewards.

To incentivize stability and long-term participation, the network offers an accrued-time bonus. This bonus is calculated as a function of the duration for which a node has been active and continuously running.

Home
Run a Node
Worker Node Setup
Running a Worker Node
Current Boot Nodes
The following nodes are used to boot into the network:

/ip4/146.190.197.136/tcp/31783/p2p/12D3KooWMUFNmnPBEZY5y7qqB8F6eVEzfPXG7KAd1v1FV1Q44A6d
/ip4/209.38.5.92/tcp/30564/p2p/12D3KooWK2qKNvmuYeQ7TFSkja8wqgSdvpfEYKgqkvRSf1GtpHEN

Prerequisites
Before you begin, ensure you have the following prerequisites:

A machine with Docker installed and running.

Optional: Valid AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY for an S3-compatible storage provider if you choose to back up your node's keys and configuration to S3-compatible storage.
A specified KEY_PATH and KEY_PASSWORD for accessing your backup data, if using S3-compatible storage.
Alternative Backup Options
If you are not using S3-compatible storage, you can back up your data using Kubernetes or a Docker-managed persistent disk. Here are some general steps to consider:

Kubernetes: Utilize persistent volumes in your pod specifications to ensure data persists across pod restarts and deployments. Consider using StatefulSets for applications that require stable, unique network identifiers.

Docker: Configure volume mappings in your Docker commands to link container storage to a persistent disk on the host system. Ensure the host's disk is backed up regularly using your preferred system backup tools.

Running the b7s Docker Image
Run the Worker Node
While b7s is multi-platform, we currently offer x86_64 enabled docker images, arm64 images are on the way.

Worker Node Configuration with AWS Backup
To run the node as a worker using AWS for backups, configure the Docker container with the environment variables for AWS and port mappings. Use the following command, replacing the placeholders with your actual values:

AWS_ACCESS_KEY_ID this is the KEY you need for an S3 Bucket
AWS_SECRET_ACCESS_KEY this is the secret generated for the S3 Access Key
KEY_PATH this is the Bucket Path, typically just the bucket name
KEY_PASSWORD this is a chosen password used for PGP/GPG encryption of the node identity information before storing in S3, make a backup this, it is used to restore the identity if the container is stopped.

```
docker run --platform=linux/amd64 -d \
-e NODE_ROLE=worker \
-e AWS_ACCESS_KEY_ID=${YOUR KEY} \
-e AWS_SECRET_ACCESS_KEY=${YOUR SECRET} \
-e KEY_PATH=${YOUR BUCKET PATH} \
-e KEY_PASSWORD=${YOUR BACKUP PASSWORD} \
-e S3_HOST=https://s3.us-west-1.amazonaws.com \
-e
BOOT_NODES=/ip4/146.190.197.136/tcp/31783/p2p/12D3KooWMUFNmnPBEZY5y7qqB8F6eVEzfP
XG7KAd1v1FV1Q44A6d,/ip4/209.38.5.92/tcp/30564/p2p/
12D3KooWK2qKNvmuYeQ7TFSkja8wqgSdvpfEYKgqkvRSf1GtpHEN \
ghcr.io/blocklessnetwork/b7s:v0.6.2
```

Worker Node Configuration with Local Key Path Mount
To run the node as a worker using a local or Docker-managed mount for key storage, omit the AWS environment variables and map the key storage directory directly. Use the following command, replacing the placeholders with your actual paths:

```
docker run --platform=linux/amd64 -d \
-e NODE_ROLE=worker \
-e
BOOT_NODES=/ip4/146.190.197.136/tcp/31783/p2p/12D3KooWMUFNmnPBEZY5y7qqB8F6eVEzfP
XG7KAd1v1FV1Q44A6d,/ip4/209.38.5.92/tcp/30564/p2p/
12D3KooWK2qKNvmuYeQ7TFSkja8wqgSdvpfEYKgqkvRSf1GtpHEN \
-v /path/to/your/local/keys:/app/keys \
-v /path/to/your/local/db_path:/app/db \
-p 9527:9527 \
ghcr.io/blocklessnetwork/b7s:v0.6.2
```

Additional Configuration
Volume Configuration
The command includes volume mappings that map directories on your host machine to the Docker container. These mappings are used to persist the database data and, optionally, key storage used by your b7s node. Ensure the directories exist on your host or Docker will create them for you, which might not have the desired permissions.

Network Configuration
The port 9527 is mapped from the host to the container (-p 9527:9527). This is the peer-to-peer (P2P) communication port. Ensure that this port is open in your firewall settings to allow incoming and outgoing connections to and from other nodes.

Monitoring and Logs
To monitor the logs of your b7s worker node, use the Docker logs command:

```
docker logs -f b7s-worker-aws
docker logs -f b7s-worker-local
```

These commands will follow the log output in the terminal, allowing you to monitor the operational status of your worker node in real-time.

Home
Run a Node
Register AVS
Register Blockless Worker as an AVS node.
Prerequisites
Docker v24 and above
EigenLayer CLI
This guide is broken down into two parts.

Setting up an EigenLayer Operator
Opt In / Opt Out the Operator for Blockless AVS
You may skip part 1 if you already have an Operator registered with EigenLayer Holskey.

Setting up an EigenLayer Operator
Follow EigenLayer guide and Install EigenLayer CLI(opens in a new tab)
Generate ECDSA and BLS keypair using the following command

```
echo "password" | eigenlayer operator keys create --key-type ecdsa [keyname]
echo "password" | eigenlayer operator keys create --key-type bls [keyname]
```

Please ensure you backup your private keys to a safe location. By default, the encrypted keys will be stored in ~/.eigenlayer/operator_keys/

Fund at least 0.3 ETH to the ECDSA address generated. It will be required for node registration in the later steps. Obtain Holskey Funds https://holesky-faucet.pk910.de/(opens in a new tab)

Register on EigenLayer as an operator
Create the configuration files needed for operator registration using the following commands. When completed, operator.yaml and metadata.json will be created.

```
eigenlayer operator config create
```

Edit the metadata.json, with the details of your new operator, this can be hosted on gist.github.com(opens in a new tab), make sure to set as a public gist. Then get the RAW url.
https://gist.githubusercontent.com/${USER}/${HASH}/metadata.json(opens in a new tab)

```
{
  "name": "Example Operator",
  "website": "<https://example.com/>",
  "description": "Example description",
  "logo": "<https://example.com/logo.png>",
  "twitter": "<https://twitter.com/example>"
}
```

Upload metadata.json to a public URL. Then update the operator.yaml attribute metadata_url with the url to metadata.json.

Execute the following command, to register the operator.

```
eigenlayer operator register operator.yaml
```

If the previous command was successful the following output should be seen in the terminal

✅ Operator is registered successfully to EigenLayer

To update this metadata in the future you can run

```
eigenlayer operator update operator.yaml
```

After your operator has been registered, it will be reflected on the EigenLayer operator page. Testnet: https://holesky.eigenlayer.xyz/operator(opens in a new tab)

The status of an operator can be checked using

```
eigenlayer operator status operator.yaml
```

Opt In / Opt Out the Operator for Blockless AVS
Create the Environment File blockless-operator.env for Docker. This allows seamless passing of options for Opt-In and Opt-Out.

```
# modify these for your keys and operator
METADATA_URI=https://path/to/metadata.json
OPERATOR_BLS_KEY_PASSWORD=
OPERATOR_ECDSA_KEY_PASSWORD=
# can leave
USER_HOME=/app
EIGENLAYER_HOME=/app
NODE_ECDSA_KEY_FILE_HOST=/app/operator_keys/opr.ecdsa.key.json
NODE_BLS_KEY_FILE_HOST=/app/operator_keys/opr.bls.key.json
```

Create the operator configuration file operator.config.yaml, this will be used in the registration step, and later by the AVS

```
production: false

# change this to your operator address
operator_address: 0x8FE5235d4c6A6c0a47FaE67E58034390a772a1da

# below can be left
avs_registry_coordinator_address: 0x2513c8b9c7c021A73F34f18DccFDA15d462d7cD7 #
registryCoordinator
operator_state_retriever_address: 0x055733000064333CaDDbC92763c58BF0192fFeBf #
operatorStateRetriever
eth_rpc_url: https://ethereum-holesky-rpc.publicnode.com
eth_ws_url: wss://ethereum-holesky-rpc.publicnode.com
ecdsa_private_key_store_path: /app/operator_keys/opr.ecdsa.key.json
bls_private_key_store_path: /app/operator_keys/opr.bls.key.json
aggregator_server_ip_port_address: localhost:8090
```

```
eigen_metrics_ip_port_address: localhost:9090
enable_metrics: true
node_api_ip_port_address: localhost:9010
enable_node_api: true
register_operator_on_startup: false
token_strategy_addr: 0x80528D6e9A2BAbFc766965E0E26d5aB08D9CFaF9
avs_service_manager_addr: 0xFb309198FEf7Ea7BC1c1d10c0E7A37A0549EECc1 #
BlocklessAVSServiceManager
```

Opt-In to the Blockless AVS by running the following command, assuming Docker is available.

```
  docker run --env-file /path/to/blockless-operator.env \
  --rm \
  --volume "/path/to/opr.ecdsa.key.json":/app/operator_keys/opr.ecdsa.key.json \
  --volume "/path/to/opr.bls.key.json":/app/operator_keys/opr.bls.key.json \
  --volume "/path/to/operator.config.yaml":/app/operator.config.yaml \
  ghcr.io/blocklessnetwork/blockless-avs-tools:v0.0.3 \
  --config /app/operator.config.yaml \
  register-operator-with-avs
```

Opt-Out of the Blockless AVS By Running the following command, assuming Docker is available.

```
  docker run --env-file /path/to/blockless-operator.env \
  --rm \
  --volume "/path/to/opr.ecdsa.key.json":/app/operator_keys/opr.ecdsa.key.json \
  --volume "/path/to/opr.bls.key.json":/app/operator_keys/opr.bls.key.json \
  --volume "/path/to/operator.config.yaml":/app/operator.config.yaml \
  ghcr.io/blocklessnetwork/blockless-avs-tools:v0.0.3 \
  --config /app/operator.config.yaml \
  deregister-operator-with-avs
```

Home
Developer Tools
Overview
Developer Tools Overview
Learn how to use the Blockless Dashboard, CLI, and developer SDKs to help you build, test, and manage your Blockless applications.

Blockless Dashboard
The Blockless Dashboard is a powerful web-based interface that allows you to manage and monitor your applications, resources, and accounts on the Blockless Network. With an intuitive user interface, you can easily access various features such as usage monitoring and application deployment. The dashboard is designed to give you a comprehensive view of your application's performance, enabling you to make informed decisions and adjustments as needed.

Key Features
Project management: View and manage your Functions, Triggers, or Oracles projects in one location.
Request monitoring: Monitor incoming requests to your projects with real-time data on project performance. Track execution or request errors through logs, helping you quickly identify and resolve any issues that may arise.
Access control: Set up permissions for your projects and APIs.
On-chain billing: Manage your account and project billing, and set up on-chain account abstraction helpers for fund custodianship.
Blockless CLI
The Blockless Command Line Interface (CLI) is a powerful and versatile tool that enables you to deploy and manage your applications directly from the command line. This is especially useful for developers who prefer working in a terminal environment or want to integrate Blockless into their existing development workflows. The CLI supports a wide range of commands that allow you to perform various tasks, such as deploying applications, managing resources, and

monitoring usage.

Key Features
Simple command structure: Easy-to-understand commands for managing applications and resources.
Cross-platform support: Works on Windows, macOS, and Linux environments.
Scripting and automation: Easily integrate Blockless into your existing development workflows.
Extensive documentation: Comprehensive command reference and usage examples.
Developer SDKs
To help you integrate Blockless into your applications, we provide Software Development Kits (SDKs) for various programming languages and platforms. These SDKs are designed to simplify the process of interacting with the host environment and various internal and external extensions, enabling you to develop applications more quickly and efficiently. Our SDKs currently support AssemblyScript and Rust, with more languages coming in the near future.

Key Features
Language support: SDKs available for popular programming languages and platforms.
Easy integration: Simplify the process of integrating Blockless into your applications.
Comprehensive documentation: Detailed guides, API references, and example code.
Ongoing support: Regular updates and improvements to ensure compatibility and performance.

Home
Developer Tools
Dashboard
Dashboard Overview
Blockless Dashboard is a developer-friendly, web-based tool designed to help you to monitor and manage your Blockless projects.

Key Features
Project Management
View and manage your Functions, Triggers, or Oracles projects in one location.

Request Monitoring
Monitor incoming requests to your projects with real-time data on project performance. Track execution or request errors through logs, helping you to quickly identify and resolve any issues that may arise.

Access Control
Set up permissions for your projects and APIs.

On-Chain Billing
Manage your account and project billing, and set up on-chain account abstraction helpers for fund custodianship.

Home
Developer Tools
CLI
Quick Start
Get started with the Blockless CLI
The Blockless CLI is a command line tool that makes it simple to use the Blockless Network and manage your apps.

With the Blockless CLI, you can connect to the network with your on-chain identity, quickly set up a local worker environment, and build, test, deploy, and monitor your projects right away.

Install the Blockless CLI
With curl:

```
sudo sh -c "curl -sSL
https://raw.githubusercontent.com/BlocklessNetwork/cli/main/download.sh | bash"
```

Or with wget:

```
sudo sh -c "wget
<https://raw.githubusercontent.com/BlocklessNetwork/cli/main/download.sh> -v -O
download.sh; chmod +x download.sh; ./download.sh; rm -rf download.sh"
```

To install on Windows, go to the releases page(opens in a new tab) on GitHub and download the x86 version of the Blockless CLI. Currently, the Windows ARM64 version is not supported.

Usage
To use the BLS CLI, open a terminal and run $ bls followed by the command you want to use. The command structure is as follows:

```
bls [command] [subcommand]
```

For example, to connect to the Blockless Network, you can run the $ bls login command:

```
bls login
```

Alternatively, you can use the $ bls function init command to initialize a new local project:

```
bls function init
```

Help
To see a list of available commands, you can run the $ bls or $ bls help command:

```
bls help
```

You can also use the -h or --help flag after any command or subcommand to display usage information. For example:

```
bls function -h
bls function init -h
```

Top level commands
The Blockless CLI provides a range of commands for managing your account, local components, and projects. For detailed reference, please visit the Blockless CLI Reference(opens in a new tab).

Below is a list of commonly used commands:

bls help: Displays information and usage instructions for the Blockless CLI and its available subcommands.
bls console: Opens the Blockless console, a web-based interface for managing your deployments and projects on the Blockless Network.
bls login: Authenticates and logs in to the Blockless Network using your wallet keypair.
bls whoami: Shows information about your current identity on the Blockless Network, including your public key.
bls components: Manages your local environment components, including the local worker agent and orchestrator agent. Note: If this is your first time building and testing your function, you need to use the $ bls components install command to install the local runtime components (which include the Blockless WASM Runtime Environment and the b7s networking module).

bls function: Build, test, and manage your projects and functions.
Glboal flags
Other than the help (-h or --help) global flag, there are two more flags that you can use globally.

-yes flag
You can use -y or --yes flag to set all options to the default value. For example:

```
bls function deploy -y
```

-version flag
v or -version flag can be used to verify the current version of the Blockless CLI:
```
bls -v
```

Home
Developer Tools
SDKs
Overview
Overview
The Blockless SDK offers libraries and tools for interacting with Blockless extensions and services. The SDK includes support for various programming languages, allowing you to work with your preferred language and create seamless, scalable, and secure dApps on top of the Blockless Network.

Key Features
Cross-platform compatibility: The Blockless SDK is designed to work across multiple platforms, including web, mobile, and desktop applications.
Language support: SDKs available for popular programming languages and platforms.
Easy integration: Simplify the process of integrating Blockless into your applications.
Comprehensive documentation: Detailed guides, API references, and example code.
Ongoing support: Regular updates and improvements to ensure compatibility and performance.

Home
Developer Tools
SDKs
AssemblyScript SDK
AssemblyScript SDK Overview
Blockless AssemblyScript SDK Overview
The Blockless AssemblyScript SDK is a set of internal extensions that provides developers with a convenient way to build Blockless applications that interact with various system resources.

The SDK includes several extension modules that provide bindings to common resources, including HTTP, JSON, Memory, IPFS, AWS S3, and CGI.

Extension Overview
HTTP: The http extension provides bindings for sending HTTP requests and receiving HTTP responses. It allows developers to make HTTP requests to remote servers and handle the resulting data in their Blockless applications.
JSON: The json extension provides bindings for encoding and decoding JSON data. It allows developers to easily serialize and deserialize JSON data in their Blockless applications.
Memory: The memory extension provides a way to access data directly from host’s memory. It allows developers to directly pass secrets via the networking layer and access the secrets within the runtime environment.
IPFS: The ipfs extension provides bindings for interacting with the InterPlanetary File System (IPFS), a peer-to-peer network for storing and sharing files. It allows developers to add and retrieve files from IPFS in their

Blockless applications.
AWS S3: The s3 extension provides bindings for interacting with Amazon S3, a cloud storage service. It allows developers to upload and download files from S3 in their Blockless applications.
CGI: The cgi extension provides bindings for accessing extensions written in other languages via the Common Gateway Interface (CGI).
Usage
You can start using the Blockless AssemblyScript SDK by installing the package to your new or existing project.

Using npm
```
$ npm i @blockless/sdk
```

Using yarn
```
$ yarn add @blockless/sdk
```

Home
Developer Tools
SDKs
AssemblyScript SDK
Memory
Memory Extension
Blockless Assembly Script SDK’s memory extension provides a way for you to read pre-set environment variables or any data passed from the networking layer directly from host memory.

Below is an example(opens in a new tab) of using the memory extension with Assembly Script:

```
import "wasi";
import { Console } from "as-wasi/assembly";
import { memory } from "../assembly";

let stdin = new memory.Stdin().read().toJSON();
if (stdin) {
  let results = stdin.get("results");
  if (results) {
    let newValue = Number.parseFloat(results.toString()) + 1000;
    Console.log("Hello " + newValue.toString());
  }
}

let envVars = new memory.EnvVars().read().toJSON();
if (envVars) {
  let environmentValue = envVars.get("ENV_VAR_NAME");
  if (environmentValue) {
    Console.log("Hello " + environmentValue.toString());
  }
}
```

For detailed reference, please refer to the reference section.

Module Install
You can start using the Blockless AssemblyScript SDK by installing the package to your new or existing project.

Using npm
```
$ npm i @blockless/sdk
```

Using yarn
```
$ yarn add @blockless/sdk
```

Module Import
```
import { memory } from "@blockless/sdk";
```

Usage

The memory extension module provides two classes, Stdin and EnvVars, to read data from host memory.

Stdin class

Stdin allows you to read data passed from the networking layer directly. To read data from Stdin, create a new instance of the Stdin class and use the read() method followed by toJSON() to parse the data into a JSON object.

You can also use toString() method to parse the data into a string object.

```
import { memory } from "@blockless/sdk";

let stdin = new memory.Stdin().read().toJSON();

if (stdin) {
  // Access data from the parsed JSON object
}
```

EnvVars class

EnvVars provides access to pre-set environment variables. Create a new instance of the EnvVars class and use the read() method followed by toJSON() to parse the environment variables into a JSON object.

You can also use toString() method to parse the data into a string object.

```
import { memory } from "@blockless/sdk";

let envVars = new memory.EnvVars().read().toJSON();

if (envVars) {
  // Access environment variables from the parsed JSON object
}
```


HTTP Extension

The Blockless Assembly Script SDK's HTTP extension allows you to make HTTP requests and handle HTTP responses directly from your AssemblyScript code. It provides a convenient way to communicate with external APIs and services.

Here is an example(opens in a new tab) of using the HTTP extension with AssemblyScript:

```
import 'wasi'
import { Console } from 'as-wasi/assembly'
import { json, http } from '@blockless/sdk'

let handle: http.HttpHandle | null = http.HttpOpen(
      'http://httpbin.org/anything',
      new http.HttpHandleOptions('GET')
)

if (handle != null) {
      Console.log(`code:${handle!}`)
      Console.log(handle!.getHeader('Content-ype')!)
      let body = handle!.getAllBody()!
      Console.log(`${body}`)
```

```
      let jsonObj = <json.JSON.Obj>json.JSON.parse(body)
      let kvs = jsonObj.valueOf()
      if (kvs != null) {
            let keys = kvs.keys()
            for (let i = 0; i < keys.length; i++) {
                  let key = keys[i]
                  Console.log(`${key}: ${kvs.get(key)}`)
            }
      }
      handle!.close()
}
```

Module Install
You can start using the Blockless AssemblyScript SDK by installing the package to your new or existing project.

Using npm
```
$ npm i @blockless/sdk
```

Using yarn
```
$ yarn add @blockless/sdk
```

Module Import
To import the HTTP extension, add the following line to your AssemblyScript file:

```
import { http } from '@blockless/sdk'
```

Home
Developer Tools
SDKs
Typescript SDK
Typescript SDK Overview
Blockless TypeScript SDK Overview
The Blockless Javy SDK for WebAssembly (WASM)! This SDK is designed to provide a seamless development experience for building, deploying, and interacting with WebAssembly modules using Javy, a lightweight JavaScript runtime powered by WASM.

Lightweight and Fast: Execute JavaScript code within a minimal WASM runtime, optimized for speed and efficiency.
Secure Execution: Run untrusted code safely in a sandboxed environment, reducing the risk of vulnerabilities.
Cross-Platform Compatibility: Build once and run anywhere—on servers, in browsers, or on embedded systems.
Rich API Support: Access a comprehensive set of APIs for working with WASM modules, including module loading, function invocation, and memory management.
Developer-Friendly: Enjoy a straightforward development experience with clear documentation, examples, and tooling support.
Vanilla JavaScript Compatibility
One of the standout features of the Javy SDK is its focus on pure, vanilla JavaScript. Javy is designed to execute standard JavaScript code without relying on Node.js-specific APIs or dependencies. This means that any JavaScript code or libraries that compile to vanilla JavaScript can be seamlessly packaged and run within the Javy runtime.

Why Javy?
By focusing on vanilla JavaScript, Javy provides a versatile and portable environment that enables developers to:

Package Existing Code: If you have JavaScript code that adheres to standard ECMAScript (ES) syntax and functionality, it can be easily compiled to WebAssembly and executed with Javy.

Avoid Dependencies: Javy doesn’t require Node.js APIs, which means you can run your JavaScript code in a minimal, resource-efficient environment. This is ideal for environments where installing and managing Node.js isn’t feasible or necessary.
Maximize Portability: Since Javy adheres strictly to vanilla JavaScript, your code can run consistently across different platforms, from browsers to edge devices, without modification.
Getting Started
Follow this instructions to setup the Blockless CLI, the developer experience tooling for building and deploying to the Blockless Network. The setup instructions can be found here