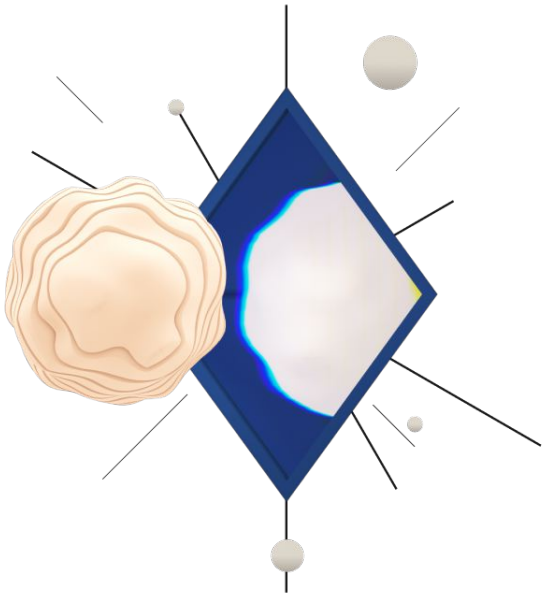
Inverter Protocol

Technical Specification



Version 2.0

August 2024

Table of Contents

Abstract

The Inverter Protocol presents a groundbreaking approach to token programmability in the blockchain ecosystem. This technical specification outlines a modular, flexible, and secure framework designed to support a wide range of tokenization use cases, from decentralized finance to real-world asset tokenization. At its core, Inverter employs a modular architecture centered around a core orchestrator contract, enabling seamless integration of various modules and existing protocols.

Key features include a governed module library, ensuring security and adaptability, and a sophisticated deployment structure utilizing the beacon proxy pattern for efficient upgrades and gas optimization. The protocol's versioning system allows for smooth updates while maintaining backward compatibility, with built-in mechanism to stop affected contracts and allow for a safe retrieval of funds.

The architecture of the Inverter Protocol comprises essential modules such as the Authorizer, Funding Manager, and Payment Processor, alongside customizable Logic Modules. This structure facilitates the creation of complex token economies, including Primary Issuance Markets (PIMs) for dynamic token issuance and distribution.

The specification delves into the Inverter Protocol's commitment to interoperability and robust security measures, including timelocked upgrades and community-driven governance. It also presents example workflows demonstrating the protocol's versatility in creating tailored token ecosystems.

By providing a comprehensive toolkit for token design and management, the Inverter Protocol aims to drive innovation in collaborative finance and circular economies, making advanced tokenization accessible to a broad range of applications and industries.

Introduction

The blockchain ecosystem lacks a robust token programmability layer. Current inflexible frameworks fail to keep pace with the rapidly evolving landscape of tokenization use cases. These shortcomings stifle innovation, elevate risks, and create inefficiencies that bottleneck the transformative potential of token-based systems. Without a flexible, modular foundation, the promise of tokenization to revolutionize industries - from decentralized finance to real-world asset tokenization - remains frustratingly out of reach.

Inverter Protocol is a decentralized coordination protocol that enables programmable issuance and asset flow between parties. The protocol is designed to support any project or use case that requires hatching a token economy and defining the exchange of resources between participating entities while ensuring openness, adaptability, and ease of use. The Inverter Protocol consists of a modular architecture that seamlessly integrates different modules and existing protocols. This modular approach enables developers to create new modules that can be added to the protocol, allowing for an ever-expanding range of use cases and applications.

Inverter Protocol is built on the Ethereum Virtual Machine and leverages the latest standards and best practices in smart contract development. The protocol uses proxies for deploying new contracts. This approach slashes deployment costs and enables on-chain upgrades without requiring users to redeploy their modules, significantly enhancing flexibility and reducing operational overhead. In addition to its modularity, one of the key benefits of the Inverter Protocol is its focus on security. The protocol comprises audited and community-accepted modules that have been tested, audited, and proven to work. Moreover, the Inverter Protocol follows an open development process, allowing anyone to vet, verify, and contribute to the codebase, further enhancing the protocol's transparency and reliability.

While Inverter was initially conceived as a dynamic funding protocol for projects and contributors with multiple funders, it has become much more than that. It is now a programmable and dynamic token engine that can be used in a wide variety of use cases, from primary issuance markets over dynamic staking mechanisms to fully-fledged protocols.

A core focus of the Inverter Protocol is to enable the programmable and dynamic issuance and distribution of tokens through Primary Issuance Markets (PIMs). PIMs employ algorithms to dynamically issue tokens based on real-time data and market conditions tailored to meet specific goals and KPIs relevant to each token's custom use case. By focusing on PIMs, the Inverter Protocol aims to drive innovation in tokenizing real-world assets and credit products, unlocking new possibilities for collaborative finance and circular economies.

Inverter Protocol's modular design ensures that each component, from token issuance to utility management, can be customized independently yet interoperate seamlessly. The protocol offers a library of ready-made, upgradable, and interoperable token mechanisms,

DeFi integrations, and an algorithm library for custom pricing algorithms. In addition, our protocol stack integrates an agent library for verifiable market-making agents and a customizable no-code admin panel for streamlined tokenization through configuration, deployment, and operation of token economies.

As the Inverter Protocol continues to evolve, we strive to create a strong and unified ecosystem for token design with its robust infrastructure and open library, making the powers of tokenization accessible to a wide range of applications and industries.

Requirements

Inverter Protocol is designed to provide a flexible and extensible way for any project or protocol to issue and exchange assets between parties programmatically, with a specific focus on enabling the issuance and distribution of tokens through Primary Issuance Markets (PIMs). As such, it must meet certain requirements to ensure its effectiveness and usefulness for the wider open-source and blockchain communities, particularly in token economy-related use cases. We must clearly define these requirements to create sound technical specifications for the protocol and smart contracts.

The main goal of the technical specification is to ensure that these high-level requirements are met, even as the implementation evolves beyond its first version. By doing so, we can create a foundation that can be extended and adapted to meet the needs of different projects and use cases without requiring a complete rewrite of the underlying code while also driving innovation in the realm of collaborative finance and circular economies.

High-Level Requirements

High-level requirements are overarching principles and goals that guide the development of a system or project. They provide a clear understanding of what the system aims to achieve, its intended functionality, and the constraints it must operate within. In the context of Inverter, high-level requirements ensure that the protocol meets certain security,

compatibility, and usability standards, allowing it to be effectively integrated into a broader range of applications. These requirements are:

- **Modular Architecture**
  - The overall design strives for modularity by distinctly separating the various steps, protocols, and actors from the code's foundation. The overall functionality is divided between a core orchestrator contract and several modules that can be activated or deactivated based on the specific requirements of the use case.
  - There is a high-level interface for each module that the orchestrator contract can rely on when communicating with them. On top of that, each module should have a clear and concise interface that outlines the expected input parameters and output data, ensuring that each module can be easily understood and tested.
  - Modules follow a standardized format, so external contributors may easily create their own modules without spending too much time understanding the intricacies of our protocol.
  - The modular architecture allows for easier code-base maintenance, enabling developers to fix bugs and update features without overhauling the entire system.
- **Governed Module-Library**
  - The Inverter Protocol has a module library that lists all the available modules. A governance mechanism will maintain the library, allowing users to add, remove, or update modules based on the community's needs and use cases.
  - This process may happen through a combination of on-chain and off-chain governance mechanisms to balance decentralization and efficiency.
  - The curation of the module library prioritizes security and diligence considerations, as the integrity and safety of the system depend on a safe and well-maintained library. Any changes or additions to the module library undergo external audits.
  - In addition to auditing, we have implemented fallback mechanisms to handle potential issues such as locked funds. Currently, we utilize an emergency stop mechanism that, when activated, allows us to deploy a sunset version of the affected contract. This sunset version operates in a rescue mode, allowing users to withdraw their deposits and funds, thus avoiding negative impacts. We are actively developing a more sophisticated, automated fallback mechanism to further enhance the security and reliability of our protocol.
- **Interoperability and Compatibility**
  - The Inverter Protocol prioritizes interoperability and compatibility with existing DeFi protocols and infrastructure to maximize its potential for driving adoption and innovation across various applications.

- The modular architecture and standardized interfaces enable seamless integration with other protocols, allowing for the creation of complex, multi-protocol workflows and applications.
- The protocol adheres to industry standards and best practices to ensure compatibility with wallets, exchanges, and other ecosystem tools, reducing friction for users and developers.

- **Token Flow Management**
  - The Inverter Protocol is designed to enable seamless and secure token flows between various contracts and parties, providing a flexible foundation for a wide range of token-based use cases and applications.
  - Inverter Protocol's token flow management and coordination capabilities serve as a foundation for building advanced token economies and token-based applications, including PIMs and other dynamic issuance mechanisms, token-based governance systems, and other innovative token models.

Architecture

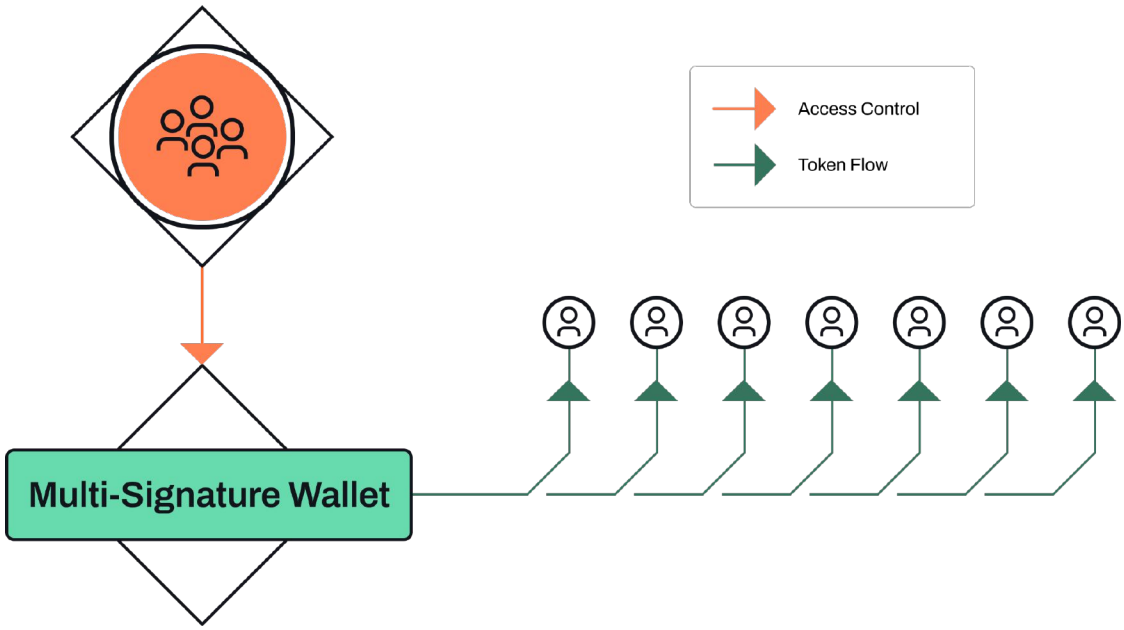
In this section, we describe the various smart contracts and their functionalities. Together with the general architecture and main interfaces, we also present the structure designed to accommodate future extensions of the overall architecture.

Overview

The goal of developing the Inverter Protocol is to create a programmable and versatile method for projects or protocols to design and operate token economies in a flexible and extensible manner. Traditionally, a central group of trusted peers designs and deploys a fixed economic design in a top-down manner. However, this approach has fundamental limitations when scaling such a system to living economies, which need to adapt and grow beyond this core group while maintaining security and flexibility.

Instead, with the Inverter Protocol, we focus on creating a system that tracks relationships and permissions among various stakeholders who may not be part of a tightly defined group. We establish clear interaction points for every party or contract without giving complete control to a single entity. Setting these boundaries allows us to extend trust beyond just one interaction and cover the entire process.

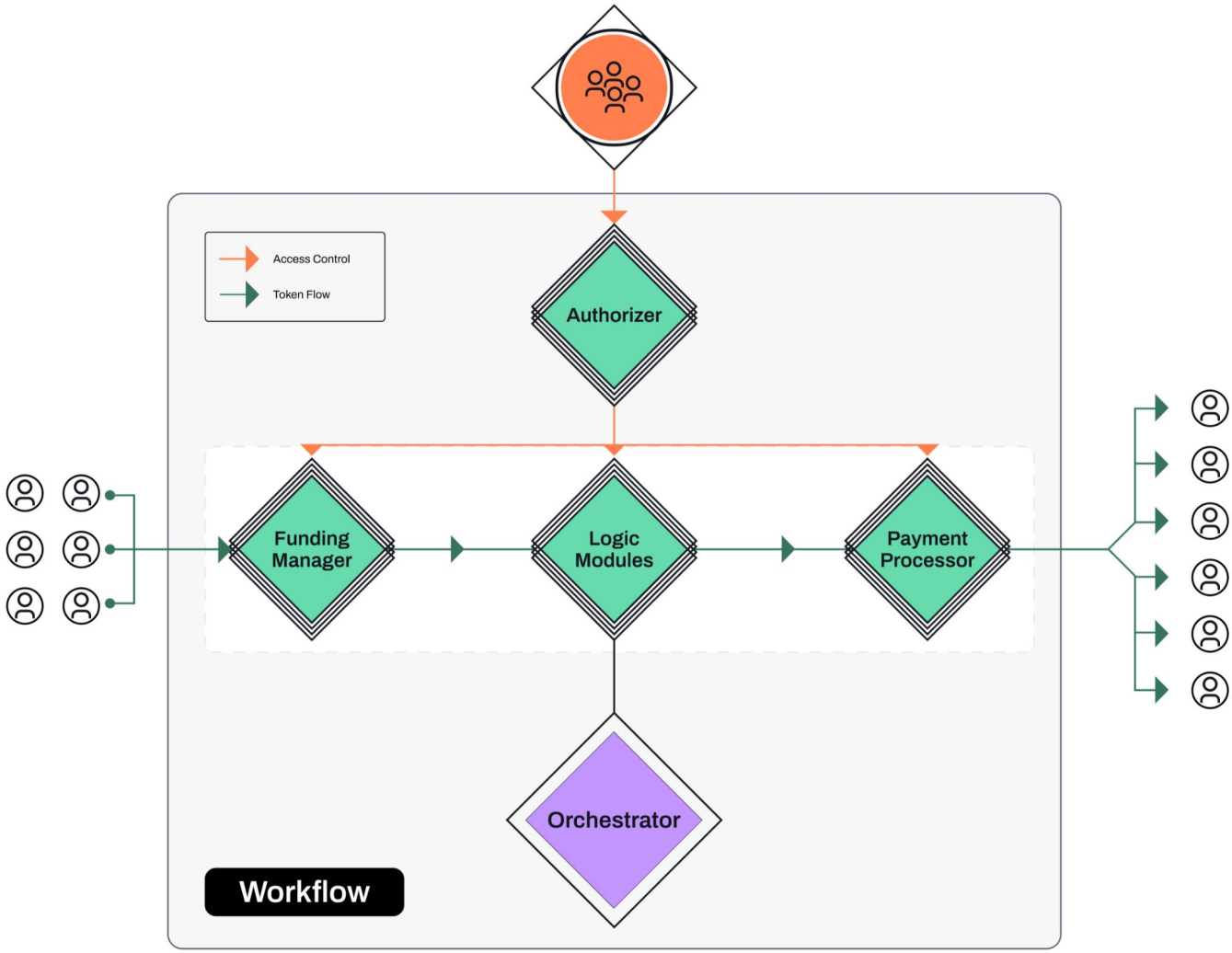
In a traditional multi-signature wallet, a trusted group of users sets rules to manage a shared pool of funds (*often using an "x-out-of-y" signature approval model*). Once the owners agree, the transfer of funds is carried out.



However, this method reaches its limits when implementing more complex flows involving multiple parties and interactions. To address this, we split the system into different parts, which are set up together and linked through a common orchestrator contract.

The Inverter Protocol's modular architecture enables dynamic token issuance and distribution paradigms, such as the Primary Issuance Market (PIM) model. The **Funding Manager** module is crucial in this process, allowing users to deposit funds while enforcing specific deposit rules and issuing tokens in return. With a PIM, this process takes the form of buying and selling the issued token, with the price and supply determined by the underlying algorithms and parameters of the chosen token issuance model.

The issued tokens and the collateral can then be managed by **Logic Modules**, which perform specific tasks and enforce custom economic designs. They are governed by a group of addresses (usually the users who set up the system), with their control being defined and governed by an **Authorizer** module, handling the system's rules and the distribution of rights. When tokens are intended to leave the system, they are handled via a **Payment Processor**, where direct control is even more restricted. This modular approach allows for the creation of complex token economies that can scale and adapt to the needs of living economies while maintaining security and flexibility.



Components

Inverter's software architecture comprises a core Orchestrator contract with modules organized around it. These modules are defined within module types, defining certain fundamental functionalities of each workflow (*i.e., a Funding Manager, Authorizer, and Payment Processor*) and a broader category of Logic Modules, which covers any module adding functionality to a workflow.

The Core Orchestrator Contract

The core orchestrator contract maps out and organizes all enabled modules. Besides these functionalities, it also stores relevant metadata and, crucially, links to the modules that are responsible for the core functions of the system:

- Management of Funds
- Authorization
- Processing of Payments
- Logic Modules

Modules responsible for these functionalities must implement specific interfaces on which the system can rely.

Module Types and Interfaces

The modules used within the Inverter Protocol implement the specific business logic a user desires to apply to their contract. As we strive to set the foundation for an open and thriving marketplace of modules, we define specific types of modules and interfaces with which the orchestrator and other modules may interact. This way, someone wanting to contribute to the Inverter Protocol by creating a module (*e.g., to integrate their own protocol or to provide a new feature*) only has to ensure that the interface requirements are met.

The Authorizer

This module defines and enforces the system's permission structure. It specifies the distinct roles and permitted actions for each stakeholder. Stakeholders may be users or other modules.

The authorization system itself is based on the notion of global and module roles. While global roles (*like the overall workflow owner*) possess certain rights within all Modules and the Orchestrator itself, a module role limits the power to a specific module's functionality.

Each Authorizer implements the IAuthorizer interface, which defines one main function:

- **hasRole(role, address):** Returns whether a specific address possesses a specific role and is thus authorized to execute the action.

Building upon a two-tiered role-based system provides the most flexibility by having overarching and localized authorization levels. Additionally, the behavior of the authorization layer can be easily altered via extension modules, as the role itself just needs to be owned by an address - and this address can also be a contract. Consider the following example:

> *If the goal is to have a workflow based on voting on every action within the system, an extension can be enabled by adding it as a module to the workflow. This module then becomes the (sole) owner of each role. Now, each authorized call to any other module is routed through it. Whenever a call to another module needs to happen, a vote must be held within that extension module. Different implementations of these extensions can then define different rules on the vote itself or the handling of the voters.*

The Funding Manager

This module holds the funds of each workflow and manages user deposits and withdrawals. In addition to the basic functionality, the Funding Manager module ultimately takes the role of

a Primary Issuance Market (PIM), enabling dynamic token issuance and redemption based on predefined algorithms and funding flows. In general, this type of module offers four types of interactions:

- **Deposit:** Users may deposit a specified amount of funds, which can be used to mint new tokens in the case of a PIM.
- **Withdraw**: Users may withdraw previously deposited funds or redeem their tokens for the underlying assets in the case of a PIM.
- **Spend:** Spends funds deposited inside the FundingManager and proportionally reduces the amount depositors can withdraw. This functionality is only available for other modules, not for end-users.

While users are making use of the deposit and withdraw functionality, the other modules within a workflow interact with the spend functionality. Therefore, for use within other modules, each Funding Manager implements the IFundingManager interface, comprised of the following functionality:

- **transferToken(module, amount)**: Transfers the token held within the funding manager to a different module.

The Payment Processor

This module receives and processes payment orders from other modules. It acts as a funnel through which all the value outflows of the system can be managed. Ultimately, it implements the individual distribution policy of the workflows' token economy, handling the monetary flows that are taking place. It implements the following functions within the IPaymentProcessor interface:

- **processPayments(module)**: Fetches the payment orders of a module and processes them accordingly.
- **cancelRunningPayments(module)**: Fetches the payment orders of a module and signals the cancellation of currently active orders. It does not make assumptions about internal bookkeeping or the return of unpaid funds.

The PaymentProcessor is working based on so-called PaymentOrders. Each logic module within the Inverter Protocol can (based on its own logic) create a PaymentOrder, outlining which payments will be made and under which conditions. An example payment order the processor receives includes:

- **Recipient:** The beneficiary of the order.
- **Token:** The payment token used for the order.

- **Amount:** The amount of the payment.
- **Timing:** The timing details of the payment order, including:
  - Start: The execution date of the payment.
  - Cliff (optional): If used, the cliff of the vested payment.
  - End (optional): If used, the end of the vested payment.

Given the IPaymentProcessor interface and the base functionality of a module, including its ability to create Payment Orders, multiple module variants can be created, implementing different approaches to asset transfers. While a straightforward implementation would just unlock a certain number of tokens once the payment terms are reached *(e.g., a specific date has been reached or a milestone has been completed)*, a more complex version of this module can use streaming protocols or other solutions, leveraging the cliff and end dates within the payment order. It generally enables value flows in multiple directions, e.g., back to the funders or other stakeholders.

Logic Modules

This category includes the remaining modules in the system. They implement the individual utility policies of the workflows' token economy, defining the conditions under which certain token flows occur. Ultimately, they implement the specific business logic a developer may wish to apply to their contract, such as requesting funds from the Funding Manager, creating payment orders, or performing any other tasks they are designed to do. There are no imposed limits on what a module can do; we just need to ensure that the communication between a specific module and the orchestrator contract, as well as the other modules within a workflow, follows a predefined schema and interface.

An example of a module would be the KPI Rewarder. This module enables the creation and management of Key Performance Indicator (KPI) based reward programs for staking. Owners can define KPIs, which are a set of tranches with rewards assigned. An external asserter can trigger the posting of an assertion to the UMA Optimistic Oracle, specifying the value to be asserted and the KPI to use for the reward distribution. The workflow utilizing the KPI Rewarder collects funds via a FundingManager, which are used to reward the stakers. Once the assertion resolves, the UMA Oracle triggers a callback function, which calculates the final reward value and distributes it to the stakers via the selected PaymentProcessor.

Creating New Modules

The Web3 space is constantly evolving and redefining itself, including the Inverter Protocol. We aim to allow the Inverter community to easily integrate new functionalities and existing

technologies as modules. Because of Inverter's modular architecture and interface guidelines, creating a new module should be simple, non-destructive, and non-intrusive.

Let's consider the following example to outline the flexibility of adding new modules:

Example Case: Earning yield on idle funds

- Builders want to generate yield by depositing a portion of the idle reserve of their token economy in their workflow contract into Morpho.
- A new module is created: The Morpho Funding Module, which adheres to the IFundingManager interface:
    - Whenever someone deposits funds, bringing the reserve above a certain ratio, they automatically deposit these to Morpho.
    - Whenever funds are to be paid out, withdraw them from Morpho and send them to the user.
    - Add any generated yield to the funding pool.

The internal design can be as simple as a standard Funding Manager or implement more complex logic: from the perspective of the rest of the system, it doesn't matter.

- The module is deployed and added to the Inverter Module Factory.
- Workflows may opt to make use of this module from now on.

The Module Library

Our approach to developing our protocol focuses on creating a user-friendly and specialized library of available modules. To achieve this, we will enable an on-chain library of whitelisted modules, which can be deployed from our workflow factory. This library consists of audited and reliable modules, ensuring the highest level of trust and safety for our users.

Currently, the Module Registry is governed by a multisig comprised of members of the Inverter Network team and well-known community members to maintain a high-quality control standard. As the protocol evolves, we will implement an on-chain governance process for a decentralized decision-making system. This transition will empower the community to play a more active role in determining which modules are included in the library while still upholding the system's integrity.

To further enhance the user experience, we will publish common patterns of module configurations designed to address a wide range of use cases. With token economy templates, we will simplify the process for users to quickly access and benefit from the platform's features tailored to their unique requirements.

Deployment Structure/Versioning

Each user should be able to deploy their own individual workflow contracts as cheaply as possible and utilize any of the pre-deployed modules in Inverter's Module Library.

Our goal in creating the Inverter Protocol was to allow for a deployment mechanism that requires module contracts to be deployed only once, making their contract logic available to any further deployed contracts. Additionally, we aimed to significantly simplify the maintenance of workflows and the overall system, enabling Inverter Protocol maintainers to quickly deploy fixes to malfunctioning modules by updating the module implementation. Subsequently, any contract utilizing the updated module will automatically employ the corrected and up-to-date logic.

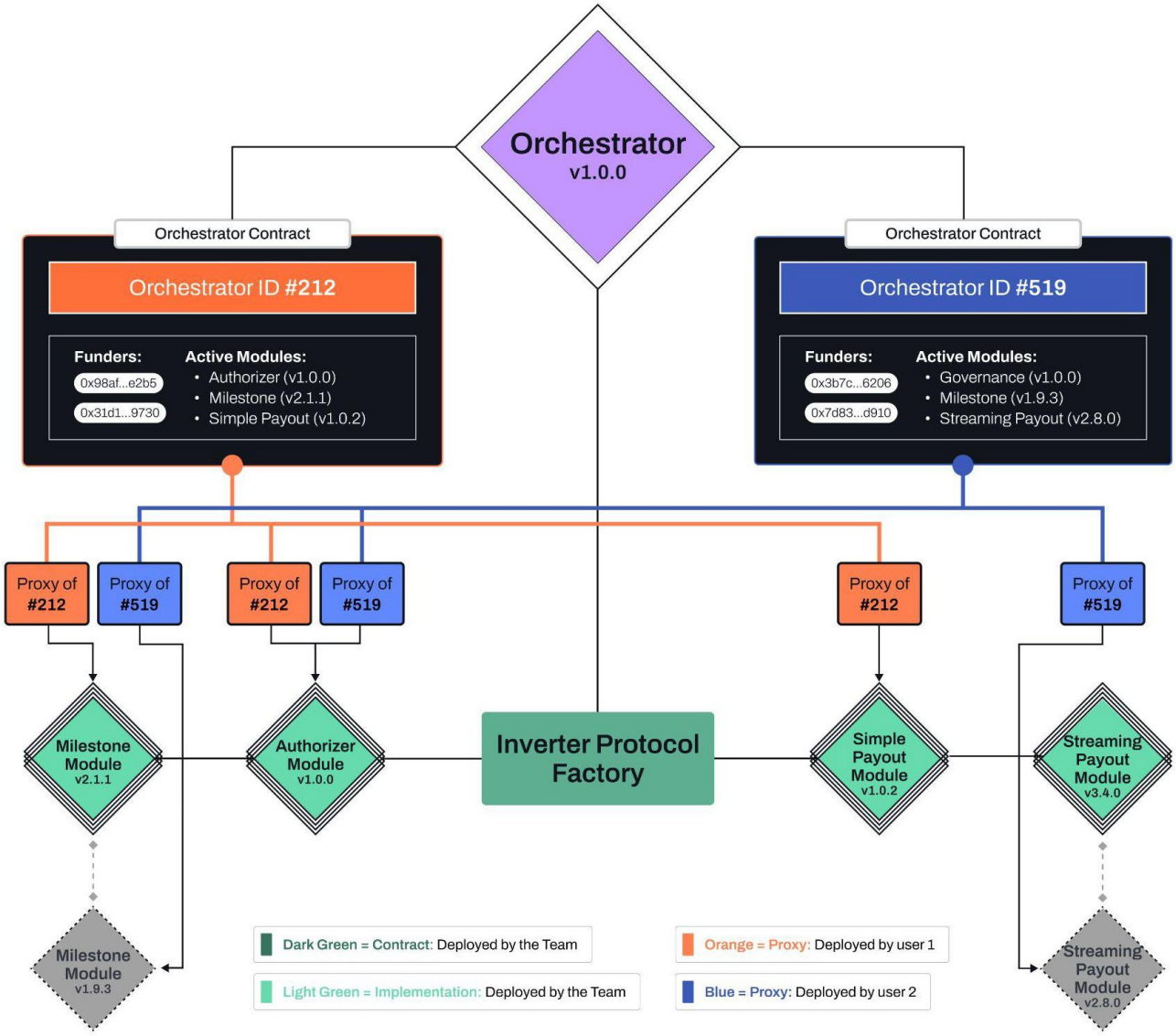
Our Proxy Pattern

We leverage the beacon proxy pattern for all of our contracts. The Beacon pattern stores the address of the implementation contract in a separate "beacon" contract. The address of the beacon is stored in the proxy contract.

With other types of proxies, when the implementation contract is upgraded, all of the proxies need to be updated. However, with the Beacon proxy, only the beacon contract itself needs to be updated. If a module ever requires an update or bug fix, then only one instance of the module needs to be updated rather than deploying an updated module for each orchestrator contract.

The appropriate multisig can set both the beacon address on the proxy and the implementation contract address on the beacon. This allows for many powerful combinations when dealing with large quantities of proxy contracts that need to be grouped in different ways and is also appropriate for situations that involve large amounts of proxy contracts based on multiple implementation contracts.

It is important to note that we employ extensive security measures to limit our own power regarding the administration of the overall beacon proxy system. A timelock mechanism, as well as a multisig composed of important community members (*and not just the maintainers of the Inverter Protocol*), ensures that there won't be any unwanted, sudden changes to a module's functionality.



Orchestrator contracts using the logic of multiple module major versions via their proxies

Independent Update Mechanism

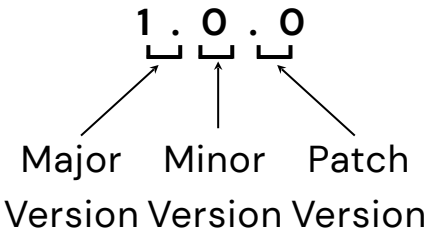
We understand that upgradable contracts, especially when the mechanism to control the upgrades themselves is not controlled by the users themselves, can raise concerns for some individuals. To address this, we have established a clear process outlining when and how upgrades can occur, as well as implemented mechanisms to limit our own power, such as a timelock that can only be bypassed if a multisig controlled by our team and community members allows it. The goal of the Inverter Protocol is to be a protocol for everyone, catering to a wide range of users with varying needs and preferences. To accommodate this, we allow our users to opt out of the automatic upgrade system by allowing them to use a regular proxy pattern during deployment. This empowers workflow owners to decide whether a specific upgrade we have deployed should be applied to their workflow or if they prefer to maintain the previous version. This flexibility ensures that users have control over

the upgrades they adopt while still benefiting from the advancements and improvements made to the protocol. For more information about this mechanism and our other security measures, please refer to our Security Guidelines.

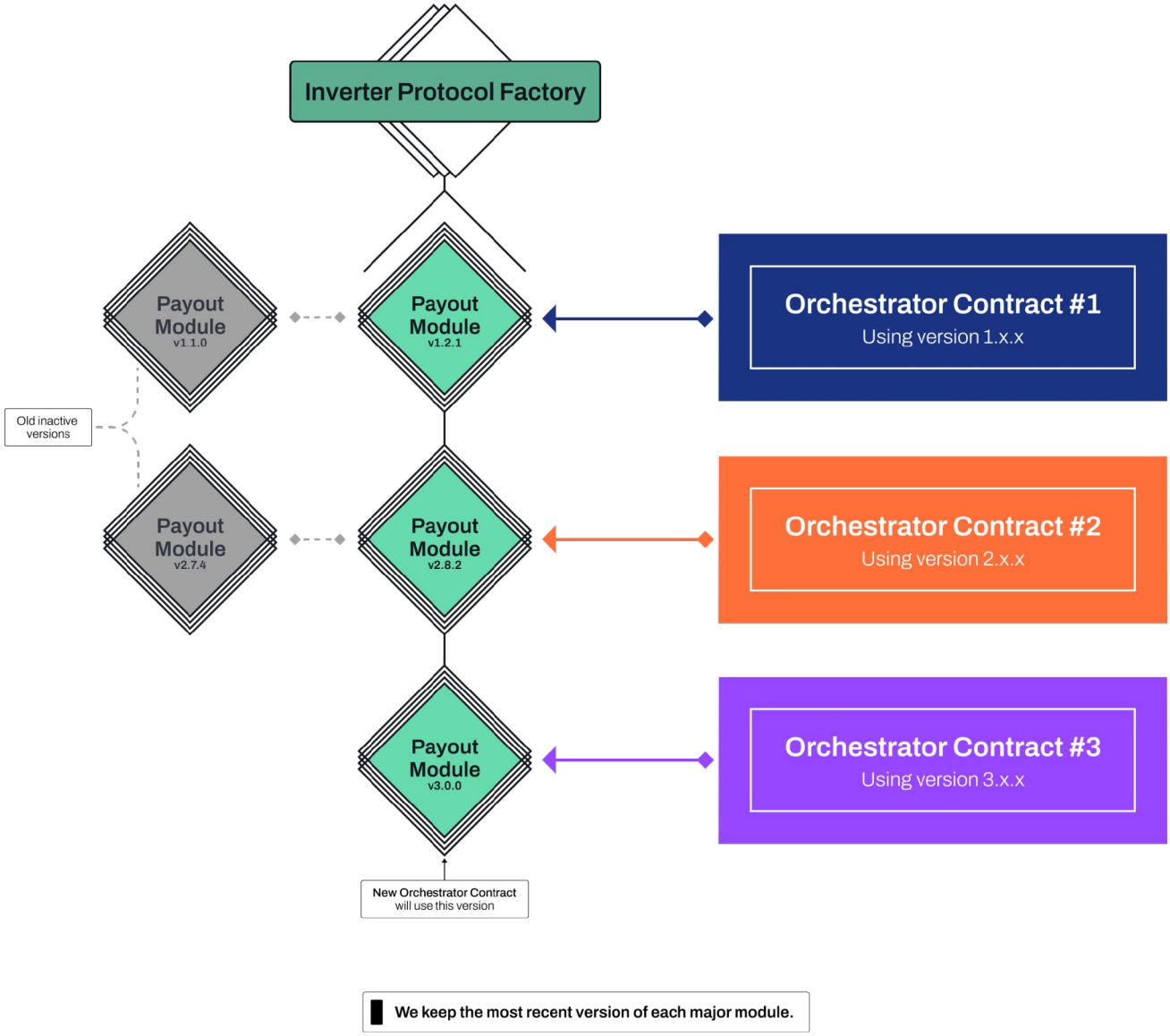
Versioning & Upgrades

There are generally two methods for updating a smart contract: a new deployment or an upgrade to the existing one. Our protocol versioning system is based on version 2.0.0 of the Semantic Versioning specification. Within our protocol, we define our versioning system based on major, minor, and patch, which are defined as follows.

- **Patch**: A change to the contract that introduces (a) non-breaking changes (*such as minor changes to its logic or modifications to the NatSpec documentation*) and (b) doesn't modify its functional interface. Full backward compatibility.
- **Minor**: A change to the contract that introduces (a) non-breaking changes and (b) modifies the functional interface while ensuring backward compatibility. The old interface still works, while the updated one unlocks added functionality.
- **Major**: A change to the contract that introduces (a) breaking changes and (b) modifies its functional interface, breaking backward compatibility.



Each time a non-breaking update to a module is done, its patch version increments if the interface remains the same. If a function was added or changed, thus modifying the interface while ensuring backward compatibility, the minor version increments, and the patch version resets to zero. Whenever a breaking update is deployed, its major version increments, and both minor and patch versions are reset to zero. The "old" major version of the corresponding module will still be available unless it is not safe anymore (see "Sunsetting Upgrade"). This way, users and developers are always aware of the modules' current state.



Within our Factory, we offer the user to choose between the latest major version of each module. If a certain module has been developed for a longer time, with its version being at 1.8.2 for example, and a completely new version is released (as 2.0.0 for example), the user may choose to either create their workflow with 1.8.2 or 2.0.0 (with the latter being the default). Users may not revert back to earlier minor versions or patch versions.

It is important to keep in mind that this system still allows us to revert to earlier, known-safe versions of the protocol if any issues are discovered in a new upgrade. This capability adds an extra layer of security and stability to our protocol, ensuring that we can quickly respond to any unexpected problems by reverting to a stable state.

Patch and Minor Version Upgrades

During a non-breaking upgrade, the major version does not change, while the patch or minor version number is generally increased by one, depending on whether the interface was

changed or not. For example, a minor module upgrade with version 1.3.2 would change its version number to 1.4.0. In this case, the contract is **automatically updated** for everyone using the beacon structure.

To allow for changes to the contracts' storage, we use the "gaps" system, where an unsigned integer array of a specific length (usually 50) is added as the last element of the storage layout of each contract. Whenever a variable is added to a contract later, its storage space is deducted from this array's length. To guarantee that a non-major version upgrade does not introduce unintended behaviors to the live system, we are using established tools to verify that the storage layout of the upgraded contract does not collide with the existing layout and that the gaps have been used correctly.

After developing a patch or minor upgrade, we extensively test the contract and upgrade mechanism on a public testnet. This allows us to test the upgrade in a controlled environment and make necessary adjustments before deploying it on the live networks.

Prior to the deployment and the actual upgrade on the live networks, a public announcement outlining the changes is made, including a link to the implementation of the deployed contract. This allows our users to familiarize themselves with the upgrade and provide feedback. At the same time, an external audit of the proposed changes will be conducted, ensuring that we did not introduce any new issues but also - if the upgrade occurs because of an issue that needs fixing - fully understand the original issue and completely resolve it with the planned upgrade. Afterward, the upgrade process, including the timelock mechanism, is initiated on-chain. During this waiting period, the Inverter Protocol users can verify the changes and provide feedback.

If the community provides negative feedback or has otherwise valid concerns, the upgrade process can be stopped, and the cycle restarted to ensure that we only implement upgrades beneficial to our users.

Once the designated duration of the timelock has passed and the feedback has been positive, the upgrade is finalized on-chain, as the beacon's implementation address is updated and pointing to the newer module.

Major Version Upgrade

If a breaking update occurs, the major version increments by one while the patch version and minor version are reset to zero. For example, an update from version 1.3.2 will change to 2.0.0 if a major version upgrade takes place. In this case, the modules used within workflows are **not automatically updated** using the beacon structure. Instead, a new module is created and registered in the module factory, which users may add to their workflow from now on.

After the development process, each new module version is deployed on a public testnet and enters the beta phase. During this time, the module is tested in a controlled environment, and necessary adjustments can be made safely before deploying it on live networks. At the same time, an external audit of the new module version will be conducted.

Once the beta phase has been completed successfully, the module will be deployed on the live networks and subsequently registered within the module factory.

Afterward, a public announcement will outline the changes within this new version, including a link to the implementation of the deployed contract. This will allow our users to familiarize themselves with the upgrade and provide feedback.

Sunsetting Upgrade

In the rare occurrence that a bug is discovered that can not be fixed via a patch or minor upgrade, we make use of a process that we call a “sunsetting upgrade”. If the resolution of the underlying bug can not be done without incurring breaking changes to the functional interface of the contract, upgrading the contract on-chain would lead to severe issues within the applications leveraging its interface and functionalities, which we do not have full control over and want to prevent at all costs.

To mitigate this and to adhere to our versioning system, we adhere to the following process:

1) The identified vulnerability within a contract is assessed in terms of the changes that are required to resolve it.
2) If it is found that the fix will introduce breaking changes, we will assess whether there are funds within that contract that require rescuing.
    a) If funds are within the contract, we create a light version of the contract that allows the respective entities that own the funds to withdraw them and blocks any other functionality of the original contract (*rescue version*).

        Example: *In the case of a staking contract, this would allow each user that has staked funds to withdraw these funds but not do anything else.*

    b) If there are no funds within the contract, we only deploy an implementation of the contract that blocks every functionality within it to prevent potential abuse.
3) The patch and minor version number of this upgrade from step (2) will be set to 99, i.e., a contract with version 1.4.2 will then become version 1.99.99, indicating that a sunsetting upgrade took place.

4) After the contract's underlying vulnerability has been fixed, we deploy it as a new contract under version 2.0.0 via a major upgrade. From that point on, users may use the contract in their workflows.

The rest of this upgrading process follows the procedures outlined for any major upgrade. The only difference is that public announcements will describe every step and clearly educate affected users on what steps they have to take to recover their funds.

Closing Remarks

As developers, we understand the importance of compatibility and interoperability when building reliable and efficient software solutions. The module library is designed to foster these qualities, but we must also acknowledge that not all integrations will be seamless. While some contracts and protocols, such as the library, can be integrated with little effort due to their natural compatibility, it is essential to note that work will be required to implement, maintain, and foster the library with integrations across the broader ecosystem.

The Inverter Protocol aims to provide the ground on which to build a diverse range of applications and economies, from tokenization verticals as base-layer blockchains and protocol tokens to community currencies, from IP-NFTs and creative work to real-world assets and tokenized invoice-based SME receivables, from micro-credit insurance pools to on-chain policy engines. While the library is a solid foundation for building more interoperable solutions, it cannot guarantee compatibility with every protocol. This is an essential constraint when working with the library, but we remain committed to providing developers with the resources they need to create groundbreaking solutions.

We can overcome challenges by working together and building a strong and unified ecosystem, such as collaborating on compatible interfaces that account for the security and functional considerations of multiple protocols. Another critical element to consider when working with the library is the reliance on the community. We recognize the importance of fostering a strong community around the library, and we are committed to working with developers, users, and other stakeholders to ensure that the library is well-supported and continues to grow over time. Specific incentive mechanisms and the game theory behind the protocol will only succeed with an active community to support the library.

As the tools we design shape us in return, tokenization has the potential to embed dynamic instruments of economic policy and incentive design into assets that will become an active force in the economies people choose to live in. We can realize this future only through positive-sum collaboration. The Inverter Protocol strives to create a strong and unified ecosystem for token design with its robust infrastructure and open library to make the powers of tokenization accessible. Mechanism designers and developers can turn their token mechanisms into reusable and composable building blocks through Inverter. For Web3

protocols that are innovating towards new primitives of financial cooperation, our protocol can act as a connective layer to facilitate an interoperability standard for projects to seamlessly integrate novel web3 technologies in accordance with their unique needs. Design agencies, researchers, and data scientists can build modeling & simulation tooling into Inverter's open-sourced SDK to allow people to make sense of the powers and capabilities of the tools they have access to. Thus, we can empower builders to dare to innovate and experiment across the frontiers of tokenization applications for meaningful purposes by providing an open design space relieved from the overbearing costs and expertise.

Conclusion

The Inverter Protocol represents a significant advancement in modular token programmability and asset flow management within the blockchain ecosystem. At its core, the Inverter Protocol's modular architecture, centered around a versatile orchestrator contract, offers unprecedented flexibility and extensibility. This design, complemented by a governed module library of audited components, ensures both security and adaptability, fostering an ecosystem of reliable building blocks for diverse tokenization use cases.

The protocol's sophisticated deployment and versioning system, utilizing the beacon proxy pattern, enables efficient upgrades and gas optimization. This approach, coupled with a nuanced versioning strategy ensures smooth transitions and maintains backward compatibility. Security remains paramount, with features such as timelocked upgrades, community multisig controls, and rollback mechanisms providing robust safeguards, while still offering users the option to opt out of automatic upgrades for additional control.

The Inverter Protocol’s architecture is inherently built around dynamic token issuance and distribution paradigms, including Primary Issuance Markets (PIMs), opening new avenues for innovative token economics. The protocol's emphasis on interoperability allows for seamless integration with existing DeFi protocols and blockchain infrastructure, while its focus on gas efficiency optimizes costs for users.

By providing a foundation for creating tailored token ecosystems, as demonstrated in the example workflows, the Inverter Protocol addresses critical gaps in the current blockchain landscape. It empowers developers, projects, and communities to create sophisticated token-based applications, breaking down traditional barriers of complexity and cost. As the blockchain space continues to evolve, the Inverter Protocol is poised to play a pivotal role in shaping the future of tokenization, driving innovation in collaborative finance, circular economies, and beyond. Its true potential will be realized through the creativity and collaboration of the wider blockchain community, paving the way for novel token economies and applications that push the boundaries of decentralized systems.

*We extend our heartfelt gratitude to **Omer Demirel**, **Mehmet Tanrikulu**, **Daniel Gretzke**, **Eric Siu**, **John Shutt** and **Patrick Rawson** for their invaluable feedback and insightful reviews. Their expertise and dedication have been instrumental in refining and enhancing this technical specification.*

Appendix: Low-Level Requirements

The Low-Level Requirements of the Inverter Protocol define the specific technical functionalities and constraints that form the foundation of the system. These requirements outline important architectural decisions, security measures, and efficiency considerations that drive the protocol's implementation. While the High-Level Requirements provide a broad vision, these Low-Level Requirements offer concrete guidelines for our development. They ensure that the Inverter Protocol maintains its modular, secure, and efficient nature while providing flexibility for future expansions and improvements.

System Architecture and Security

- Each workflow is deployed as a separate smart contract instance, isolating funds and logic. This ensures that a vulnerability in one module doesn't affect others, enhancing overall system security.
- Modules must implement specific interfaces (IAuthorizer, IFundingManager, IPaymentProcessor) to ensure compatibility with the Orchestrator and other modules. This standardization allows for seamless integration of new modules and interoperability within the system.
- Workflows can be paused at the module level or entirely in case of emergencies. This granular control allows for targeted risk management without disrupting the entire system unnecessarily.

Upgrades and Versioning

- The system offers users a choice in upgrade mechanisms: they can opt for the beacon proxy structure, allowing the Inverter team to manage upgrades, or choose a transparent proxy for full control over their upgrades. This flexibility caters to different security preferences and use cases.
- Module updates follow semantic versioning (major.minor.patch), with different processes for each type of update. Patch and minor updates can be applied automatically (if the user chose the beacon structure), while major updates are treated as entirely new modules. Users can choose to adopt these new major versions in their workflows, but their existing modules remain unchanged unless they actively choose to upgrade.
- The Module Library is governed on-chain, with a rigorous process for adding, removing, or updating modules. This ensures that only thoroughly vetted and secure modules are available for use in the system.

- New modules can be created by adhering to specified interfaces and undergoing security audits. This open architecture allows for continuous innovation while maintaining system integrity.

Gas Efficiency

- Gas efficiency is a key focus in smart contract design and implementation. Best practices such as minimizing storage usage, reducing contract interactions, and utilizing proxies for deployment are employed. Additionally, advanced Solidity features like the --via-ir pipeline are used to further optimize gas usage, ensuring cost-effective operation of the protocol.

Appendix: Example Modules

Authorizer Modules

Role-based Authorizer

Type: Authorizer Module

The Role-based Authorizer module provides a robust access control mechanism for managing roles and permissions across different modules within the Inverter Protocol, ensuring secure and controlled access to critical functionalities. It is based on OpenZeppelin's AccessControlEnumerable, extending its functionality to offer fine-grained access control through role-based permissions.

Key features and functionalities:

- Implements the general Authorizer interface, which defines the core functions for managing roles and permissions.
- Allows modules to grant and revoke roles for specific addresses or multiple addresses in batches.
- Provides functions for the workflow admin to grant and revoke global roles that apply to all modules.
- Generates unique role IDs for each module by combining the module address and role identifier, ensuring role separation between modules.

Token-gated Authorizer

Type: Authorizer Module

The Token-gated Role Authorizer module extends the functionality of the Role-based Authorizer by introducing token-based access control. It enables roles to be conditionally assigned based on token ownership, allowing for dynamic permissions tied to specific token holdings.

Key features and functionalities:

- Builds on the Role-based Authorizer by integrating token-based access checks before role assignment.
- Supports both ERC20 and ERC721 tokens as qualifiers for role eligibility.
- Allows modules to set token gating for specific roles and define token ownership thresholds.
- Provides functions to grant token roles and set token thresholds for roles.

- Overrides the hasRole function to check for token ownership when a role is token-gated.

Voting-Roles Authorizer Extension

Type: Logic Module

The Voting-Roles Authorizer module is an extension of the Role-based Authorizer that facilitates voting and motion management within the Inverter Protocol. It allows designated voters to participate in governance through proposals, voting, and decision execution.

Key features and functionalities:

- Works like a logic module, extending the functionality of the Role-based Authorizer.
- Supports setting thresholds for decision-making and managing voter lists.
- Allows voters to create motions, cast votes, and execute actions based on collective decisions.
- Provides functions for adding and removing voters, setting voting thresholds, and adjusting voting duration.

Funding Managers

Bancor Redeeming Virtual Supply Funding Manager

Type: Funding Manager

The Bancor Redeeming Virtual Supply Funding Manager module enables the issuance and redemption of tokens on a **bonding curve** using a virtual supply for both the issuance and the collateral. It integrates Aragon's Bancor Formula to manage the calculations for token issuance and redemption rates based on specified reserve ratios.

Key features and functionalities:

- Supports buying and selling of tokens in exchange for an issuance token.
- Implements virtual supply adjustments for both the issuance token and the collateral token.
- Utilizes the Bancor Formula to calculate token issuance and redemption amounts based on reserve ratios.
- Allows the workflow admin to set virtual supplies and adjust reserve ratios.
- We also offer a version that allows the workflow admin to restrict user interactions to only the holders of a certain module role (whitelisting).

Deposit Vault

Type: Funding Manager

The DepositVault Funding Manager module allows users to deposit tokens to fund the workflow. It implements a simple mechanism for users to contribute funds to the system, and only allows the workflow admin to withdraw funds (*if they have not been spent via the workflow*).

Key features and functionalities:

- Allows users to deposit a specified ERC20 token into the contract.
- Allows the workflow admin to withdraw any unspent funds.

Payment Processors

Simple Payment Processor

Type: Payment Processor

The Simple Payment Processor module manages ERC20 payment processing for modules within the Inverter Protocol compliant with the ERC20PaymentClient interface. It handles payment orders from registered modules, ensuring only eligible modules can initiate payments.

Key features and functionalities:

- Implements the general PaymentProcessor interface to handle payment orders from registered modules.
- Processes payments by transferring tokens from the payment client to the order recipients.
- Tracks payments that could not be made to the recipients and allows recipients to claim these amounts later.

Streaming Payment Processor

Type: Payment Processor

The Streaming Payment Processor module manages continuous and linear streaming payment streams within the Inverter Protocol. It allows for multiple concurrent streams per recipient and provides tools to claim streamed amounts and manage payment schedules dynamically.

Key features and functionalities:

- Supports complex payment interactions, including streaming based on time for multiple clients and recipients.
- Allows recipients to claim all streams at the same time or claim for specific streams.

- Provides functions to retrieve payment order details, such as start time, cliff duration, end time, and released amounts.
- Enables the removal of payment orders for specific streams or all payments for a recipient.
- Handles error scenarios by tracking unclaimable amounts and allowing recipients to claim them later.

Logic Modules

Bounties Module

Type: Logic Module

The Bounty Manager Module provides functionality to manage bounties and process claims, allowing participants to propose, update, and claim bounties securely and transparently within the Inverter Protocol.

Key features and functionalities:

- Extends the ERC20PaymentClient to integrate payment processing with bounty management.
- Supports dynamic additions, updates, and locking of bounties by users with the "bounty issuer" role.
- Allows users with the "claimant" role to create and update claims for bounties, specifying contributor details and claim amounts.
- Enables users with the "verifier" role to verify claims and process payments to the respective contributors.
- Provides functions to retrieve information about bounties and claims, such as bounty and claim details, IDs, and contributor-specific data.

Recurring Payment Manager Module

Type: Logic Module

The Recurring Payment Manager Module facilitates the creation, management, and execution of scheduled recurring payments within the Inverter Network, allowing for systematic and timed financial commitments or subscriptions.

Key features and functionalities:

- Uses epochs to define the period of recurring payments and supports operations such as adding, removing, and triggering payments based on time cycles.

- Allows the workflow admin to add and remove recurring payments with specified amounts, start epochs, and recipients.
- Provides functions to retrieve information about recurring payments, such as payment details, IDs, and epoch-related data.
- Integrates with the ERC20PaymentClient to handle actual payment transactions using the token type stored in the FundingManager.

Staking Module

Type: Logic Module

The Staking Module provides a flexible staking mechanism for users to stake tokens and earn rewards. It extends the ERC20PaymentClient and integrates with the Payment Processor to enable the distribution of rewards to stakers.

Key features and functionalities:

- Allows users to stake tokens and earn rewards based on the staked amount and duration.
- Calculates rewards based on a configurable reward rate and the time since the last update.
- Maintains a total supply of staked tokens and keeps track of individual user balances.
    - Allows the workflow admin to set and update the reward parameters, such as the reward amount and duration.

KPI-Rewarder Module

Type: Logic Module

The KPI-Rewarder Module extends the functionality of the Staking Module by introducing a mechanism for dynamically distributing rewards to stakers based on Key Performance Indicators (KPIs). It integrates with UMA's Optimistic Oracles to enable KPI-based reward distribution within the staking manager.

Key features and functionalities:

- Allows the admin to create KPIs, which are a set of tranches with associated reward values.
- Supports both continuous and non-continuous reward distribution based on the KPI configuration.
- Enables external actors with the "asserter" role to trigger the posting of an assertion to the UMA Oracle, specifying the KPI value and the target KPI for reward distribution.

- Handles the resolution of KPI assertions, calculating and distributing rewards to stakers accordingly.

Payment Router Module

Type: Logic Module

The Payment Router Module enables pushing payments directly to the Payment Processor, allowing for seamless and efficient fund distribution within the Inverter Network. It extends the ERC20PaymentClient to integrate payment processing functionality.

Key features and functionalities:

- Allows users with the "payment pusher" role to initiate payments by providing recipient addresses, payment tokens, amounts, and timing parameters (start, cliff, and end).
- Supports both individual payments through the pushPayment function and batched payments through the pushPaymentBatched function, enabling efficient processing of multiple payments in a single transaction.
- Integrates with the Payment Processor to process the payments immediately after they are added to the payment orders.
- Provides flexibility in specifying payment timing, allowing for immediate payments (when the start variable is set to "now") or scheduled payments with specific start, cliff, and end times.
- Implements access control using the "payment pusher" role to ensure only authorized users can initiate payments.

Appendix: Example Workflows

Dynamic Token Issuance Workflow

An example workflow

The *Dynamic Token Issuance Workflow* combines the **Bancor Redeeming Virtual Supply Funding Manager**, **Streaming Payment Processor**, and **Token-gated Authorizer** modules to create a powerful and dynamic token issuance system based on the Primary Issuance Market (PIM) framework. This workflow enables the creation of a new token based on a bonding curve, allowing for dynamic price discovery and liquidity provision.

The Bancor Redeeming Virtual Supply Funding Manager serves as the core component, enabling the issuance and redemption of tokens using a virtual supply for both the issuance and the collateral. The unique parameters of the bonding curve, such as the reserve ratios and initial supply, can be customized during the workflow initialization to suit the specific requirements of the token.

The Streaming Payment Processor module facilitates the distribution of tokens to participants in a continuous and linear manner. This allows for gradual and predictable token distribution to encourage long-term engagement.

To ensure decentralized governance and decision-making, the Tokengated Authorizer module is employed. It utilizes the newly issued token as the qualifying token for assigning roles and permissions. Token holders can participate in governance activities, such as proposing and voting on changes to the system parameters or managing the allocation of funds.

By leveraging the *Dynamic Token Issuance Workflow*, projects can create their own tokenized ecosystems with built-in liquidity, price stability, native yield generation, and community-driven governance.

KPI-based Rewards and Staking Workflow

An example workflow

The *KPI-based Rewards and Staking Workflow* is an ideal solution for a project that aims to reward participants based on the project's on- or off-chain performance metrics. The workflow incorporates the **Rebasing Funding Manager**, **Simple Payment Processor**, **KPI-Rewarder**, and **Role-Based Authorizer** modules to create a comprehensive system for funding and reward distribution.

The Rebasing Funding Manager allows the project to manage its funds. It can accept funding from the project's treasury or individual users. Users can easily withdraw any unused funds through the Rebasing Funding Manager token created as part of the workflow.

To incentivize participation and align interests, the KPI-Rewarder module is integrated into the workflow. Project token holders can stake their tokens and earn additional rewards based on the performance of the project. Those rewards get calculated by verifying performance metrics through an external oracle (*for example, the UMA Optimistic Oracle*), which then triggers the appropriate reward distribution based on a set of stored KPIs. This mechanism encourages long-term commitment and active engagement from the community.

The Role-Based Authorizer module ensures that the appropriate roles and permissions are assigned to participants within the workflow. This allows for secure and controlled access to critical functions, such as posting assertions, setting KPIs, and managing the project's funds.

By combining these modules into a cohesive workflow, a project can effectively leverage the power of dynamic issuance to incentivize participation and optimize its reward payouts toward its success.