VERISENSE NETWORK: AN ACTIVE BLOCKCHAIN NETWORK FOR ON-CHAIN AGENT EXECUTION AND MULTI-AGENT ORCHESTRATION

TECHNICAL WHITE PAPER

Verisense Team
team@verisense.network

April 26, 2025

ABSTRACT

This paper introduces Verisense Network, a novel active blockchain framework with native active data I/O, representing a foundational shift in how onchain programs interact with the real world and with each other. Traditional blockchain architectures are passive by design—smart contracts cannot proactively initiate data exchange and must rely on oracles for passive data feeds, a limitation that is increasingly inadequate in the era of decentralized AGI (dAGI), where dynamic, adaptive, and interactive agentic systems are becoming critical.To address this gap, Verisense proposes a new paradigm: active blockchain systems. These systems incorporate native active data I/O, enabling onchain programs to fetch, verify, and act on external data from APIs, web servers, databases, and even other blockchains. This breakthrough allows for new applications previously unviable on passive blockchains.Verisense captures two key value blocks in the dAGI stack: (1) a foundational agentic platform for fully onchain agent hosting, execution, and discovery—interoperable with a wide range of LLM providers (both centralized and decentralized options); (2) universal multi-agent orchestration, supporting offchain-to-onchain workflows and cross-chain coordination, effectively extending the Model Context Protocol and Google's Agent2Agent standard into a decentralized context.The key contributions of this paper include a novel architecture for active blockchain systems, the integration of semantic-based and I/O-based consensus mechanisms, and the introduction of Nucleus — the core of Verisense enabling agent onchain execution and orchestration.

Keywords Decentralized AI Agent · Active Blockchain · Agent Orchestration

1 Introduction

Blockchain systems are evolving beyond static ledgers into *active* platforms capable of dynamic interaction and decision making between decentralized agents. The growing relevance of such active blockchain systems is driven by emerging applications where autonomous agents—ranging from AI-driven services to IoT devices—must collaborate, make decisions, and transact without centralized control. However, current blockchain architectures fall short when handling decentralized agent hosting, execution and orchestration. Existing systems primarily support static transactions and pre-defined smart contracts, lacking chain-native mechanisms for hosting and orchestrating complex, adaptive behaviors among multiple autonomous agents. This limitation creates a critical gap in enabling fully decentralized *agent-to-agent* collaboration and coordination on-chain. There are four fundamental challenges in realizing a blockchain framework for active agent orchestration:

- **Coordination among Autonomous Agents:** In a decentralized environment, each agent operates with its own objectives and knowledge. Without a unified coordination mechanism, agents may work at cross-purposes, leading to conflicts, inefficient resource use, or even systemic instability. Achieving coherent collaboration among heterogeneous agents remains an open challenge.

- **Ensuring Security and Privacy:** Decentralized agents need to exchange data and make decisions in an untrusted setting. Ensuring end-to-end security and privacy for agent interactions is difficult, as sensitive information might be distributed across nodes and vulnerable to interception or tampering. Advanced cryptographic techniques (e.g., fully homomorphic encryption) promise solutions but often incur prohibitive overhead, complicating practical deployment.
- **Robust Incentive Mechanisms:** A sustainable multi-agent ecosystem requires incentive mechanisms that encourage honest participation and collaboration. Inadequate or misaligned incentives can result in selfish behavior, free-riding, or reluctance to contribute computational resources. Designing robust, game-theoretically sound incentive models is critical to align individual agent goals with the collective welfare of the network.
- **Modular and Extensible Orchestration Frameworks:** Because agents may be diverse in function and operate in changing environments, the blockchain orchestration framework must be highly modular and extensible. It should allow different agent types and use cases to plug into the network seamlessly. Achieving flexibility without compromising interoperability or performance is a significant design hurdle.

To address these challenges, this paper introduces the *Verisense Network*, a decentralized agent-oriented blockchain framework. The Verisense Network is designed as an active ledger that facilitates direct agent collaboration and reliable on-chain decision-making. In particular, our approach integrates a specialized blockchain architecture for multi-agent coordination, novel consensus protocols tailored to agent-based operations, and a workflow-driven orchestration model. Together, these innovations enable a new class of blockchain applications where autonomous agents can be created, interact, and evolve in a secure and self-organizing manner. In the following, we outline the key contributions of our work.

2 Contributions

Our contributions in this paper are threefold:

1. **Verisense Network architecture for active agent collaboration:** We propose a novel blockchain architecture explicitly optimized for collaborative interactions among autonomous agents. This architecture provides native support for dynamic agent-to-agent communication and decision processes on-chain.
2. **Novel consensus schemes for agent hosting and orchestration:** We design two complementary consensus mechanisms tailored for secure, decentralized agent hosting and orchestration. These consensus schemes ensure that multi-agent workflows execute reliably and that agent states remain verifiable and tamper-proof across the network.
3. **Workflow-based decentralized agent orchestration system:** We introduce a workflow-driven orchestration framework that supports the creation and interaction of decentralized agents. This system allows developers to define complex multi-agent workflows in a modular fashion, enabling extensible and adaptive agent behavior within the blockchain environment.

3 Recall of the Verisense Network

To reflect the latest developments in the Verisense Network, we've undertaken a comprehensive revision and iteration of its architecture and functionalities. This updated version introduces significant enhancements, including the Active Blockchain paradigm, which empowers on-chain modules to autonomously initiate external service calls and schedule computational tasks without external triggers. This advancement addresses the limitations inherent in passive blockchain designs that rely solely on external transactions for state transitions.

For a detailed understanding of the previous version of the Verisense Network, readers are encouraged to consult the earlier whitepaper [2024].

3.1 Active Blockchain

The Active Blockchain paradigm extends traditional passive blockchain designs by introducing proactive on-chain execution mechanisms, overcoming the limitations inherent in passive designs, which rely solely on external transactions for state transitions. Unlike passive blockchains, Active Blockchains enable smart contracts or on-chain modules to autonomously initiate external service calls and schedule computational tasks without external triggers, integrating these interactions into the blockchain's consensus.

Protocol Model and Invocation Mechanism Formally, the Active Blockchain can be modeled as an extended state machine:

$$\mathcal{M} = \langle \Sigma, E, T, f \rangle, \tag{1}$$

where Σ represents the state space, E denotes the event set (including external transactions and internally triggered events), T represents a discrete and totally ordered timeline, and f is the state transition function. In comparison to passive blockchains, the event set E includes two additional internal event types: asynchronous external call events and timer-triggered events. Specifically, state transitions at block height $t + 1$ can be represented as:

$$S_{t+1} = f\left(S_t, \{tx_i\}_{i \in I}, \{e_j\}_{j \in J}\right), \tag{2}$$

where $\{tx_i\}$ represents external transactions, and $\{e_j\}$ comprises internally generated events. This expanded model ensures deterministic and verifiable state transitions, even with external interactions, through rigorous cryptographic proof methods such as threshold signatures or zero-knowledge proofs (ZKPs).

Time-Driven Mechanism Active Blockchains inherently support time-driven event scheduling. By integrating timing mechanisms into the consensus layer or simulating time through block height increments, contracts can autonomously set future trigger points, significantly enhancing application scheduling capabilities. Nodes validate the occurrence of scheduled events at each block interval, incorporating triggered events directly into the block execution process, thus providing deterministic and consistent execution.

Verifiability and Security Design Given the necessity for blockchain-environment interactions, Active Blockchain designs emphasize stringent verifiability mechanisms to uphold consensus security. External calls, such as HTTP requests and cross-chain data retrievals, require multiple-node validations and consensus-driven confirmations. This distributed verification model effectively detects inconsistencies introduced by malicious actors, ensuring result authenticity and integrity. Security measures are reinforced by integrating Verifiable Random Functions (VRFs) to prevent predictability in execution ordering; Fully Homomorphic Encryption (FHE)[2009] and zero-knowledge proofs[1985] for privacy-preserving computations; and cryptographic state commitments[1988] to prevent post-facto alterations.

Comparison with Passive Blockchains From a theoretical standpoint, Active Blockchains extend the conventional passive state machine into an interactive model, providing blockchain modules autonomy to initiate interactions and timely actions typically managed externally by oracles or off-chain agents. While passive chains offer simplicity and a closed-state model, Active Blockchains significantly expand functionality and composability, directly integrating external API calls, AI computations, and multi-chain operations into the blockchain framework. Despite these enhancements, Active Blockchains must meticulously address challenges related to external data consistency and malicious exploitation, relying heavily on the robust design of invocation models, timing mechanisms, and verifiable results to achieve comparable security to passive counterparts.

3.2 Verisense Architecture

The Verisense Network is structured as a decentralized and hierarchical architecture designed to facilitate secure and efficient decentralized applications. Its key components include validators, **Monadring**[2024] subnetworks, and **Nucleus** execution environments, as illustrated in Fig. 1.

At the top level, the **Verisense Network** comprises a collection of validators responsible for overall network integrity and governance. Validators are selected through a **Verifiable Random Function (VRF)**, ensuring fairness, unpredictability, and decentralization of network participation.

Validators are further partitioned into lightweight subnetworks known as **Monadrings**, each serving as an independent consensus group within the broader network. These Monadrings implement consensus mechanisms among their validator members, processing transactions, and maintaining localized state updates autonomously. Each Monadring operates independently, yet collaboratively contributes to the overall health and scalability of the Verisense Network.

Within each Monadring, decentralized execution environments called **nuclei** are deployed. Each Nucleus serves as a secure runtime capable of proactively initiating and managing diverse tasks. Leveraging Active Blockchain technology, a Nucleus can autonomously execute various instructions.

3.2.1 Monadring

Monadring is a lightweight consensus mechanism designed to attain agreement on state updates within a Verisense subnet. Inspired by token-ring network architectures, Monadring organizes a subset of Verisense validator nodes into

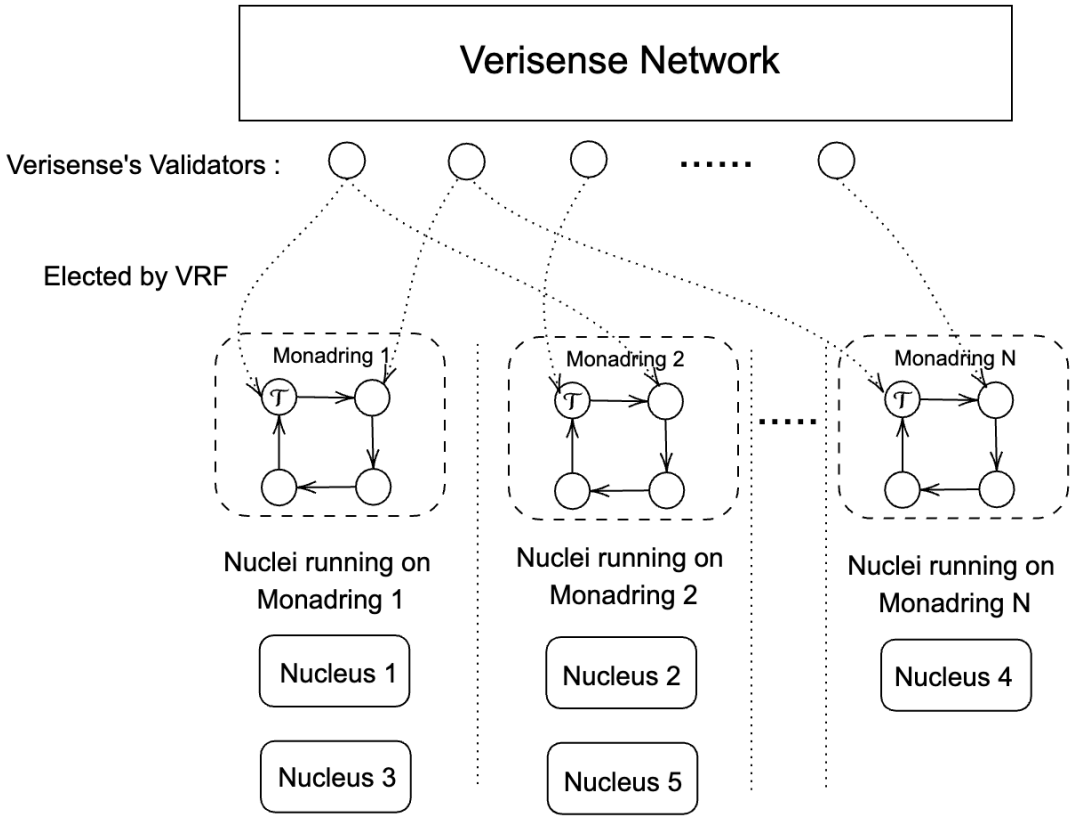

Figure 1: Illustration of Verisense Architecture

a logical ring. Formally, if $\mathcal{V}$ denotes the set of all validators on the main Verisense chain (the *Hostnet*), Monadring can partition them into multiple subnetworks $\mathcal{S}_i \subseteq \mathcal{V}$ for various applications. Each subnet $\mathcal{S}_i$ maintains its own local ledger $\mathcal{L}_i$ (distinct from the global ledger $\mathcal{L}$ of the Hostnet) to record the sequence of events and states specific to its application. The main blockchain acts as a coordination hub and registry, publicly mapping each subnet's identity to its membership and providing a verifiable record of each subnet's state (for accountability and auditing).

At the core of Monadring is a **token-passing protocol** that serves as a rotating leader election mechanism, whose simplified workflow is illustrated in Alg. 1. A single *token* continuously circulates around the ring of nodes in the subnet, and only the node currently holding the token is authorized to propose and commit state modifications. This design dramatically simplifies consensus: instead of all nodes competing to propose blocks, the token holder exclusively drives the next state transition. When a node receives the token, it processes any pending events carried by the token (these events may have been initiated by other nodes while they did not hold the token), applies the corresponding state changes, and then executes its own new events or transactions. After updating the state and appending its new events (if any) into the token (along with cryptographic attestations of the new state), the node passes the token to the next validator in the ring order. All other nodes validate the token's contents (the proposed events and resulting state) when they receive it. Once a majority of the subnet (or a predefined quorum) has validated and the token completes a full cycle, the contained batch of events and the resultant state are considered *committed* in the subnet's ledger. This guarantees a globally recognized sequence of events within the subnet, akin to a block in a traditional blockchain, but achieved with minimal overhead. The illustration of the Monadring architecture is shown in Fig. 2.

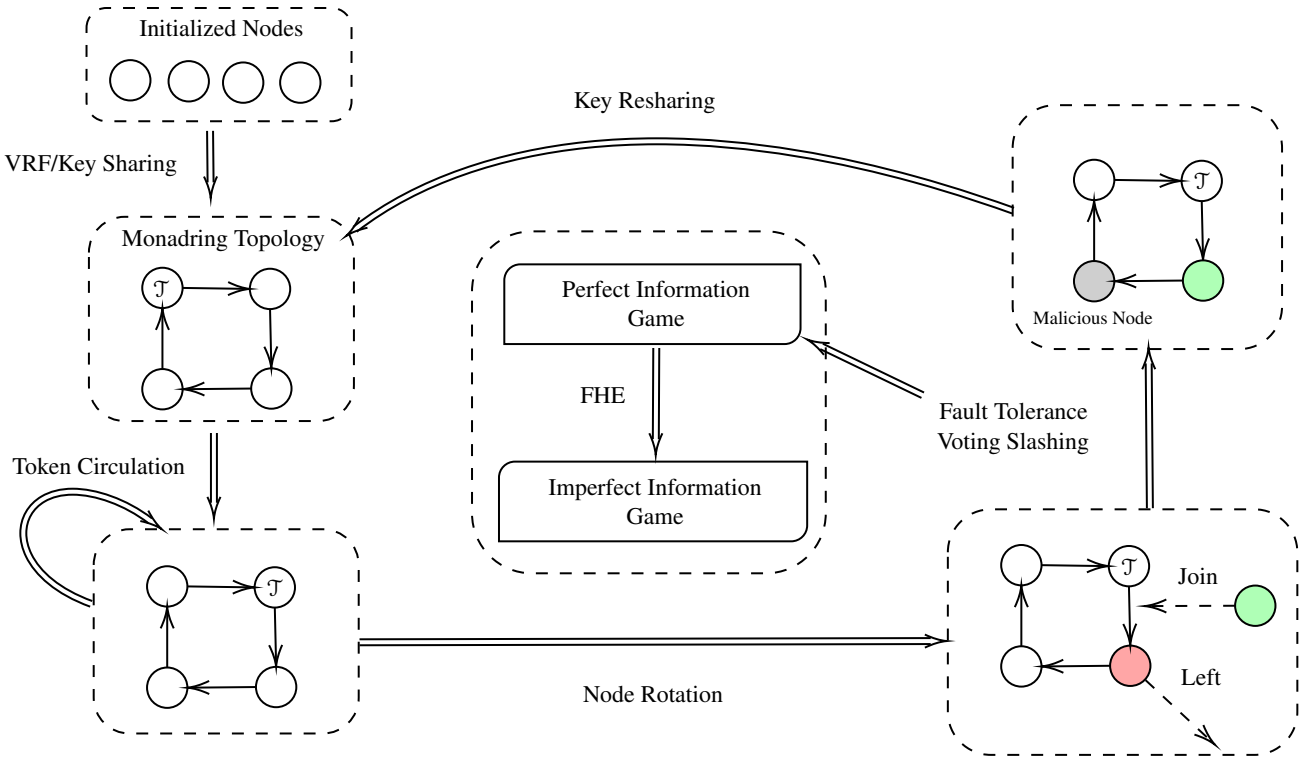

Figure 2: Overview of the Monadring Consensus Mechanism. Nodes are initially grouped and initialized via Verifiable Random Functions (VRFs) and Threshold Key Sharing, forming a logical ring topology (Monadring topology). Consensus is sequentially achieved through token circulation, where the token ($\mathcal{T}$) holder has the exclusive right to propose state updates. To mitigate strategic manipulation risks, Monadring employs Full Homomorphic Encryption (FHE), transitioning consensus from a perfect information game to an imperfect information scenario by concealing sensitive transaction details from intermediate validators. Fault tolerance mechanisms—including voting, slashing, and node rotation—ensure network robustness, handling malicious or inactive nodes efficiently. Nodes can dynamically join or leave the subnet without disrupting consensus, aided by secure key resharing procedures, thus maintaining subnet continuity and security.

Algorithm 1 Monadring Consensus Token-Passing (Simplified)

Require: Subnet members $N_1, N_2, \ldots, N_k$ arranged in a ring (order determined by VRF sorting)
Ensure: Continual agreement on an ordered log of events (ledger $\mathcal{L}_i$) for the subnet

initialize token with current state s_0 and no events
while network is active **do**
  for each node N_j in ring order 1 to k **do**
    N_j receives token (with state s and event list E)
    N_j executes all events in E on its local state, obtaining updated state s'
    N_j collects new local events E_j (if any) generated during this interval
    N_j appends E_j to token's event list and updates token's state to s''
    N_j signs the token (attesting to s'' and E_j), then passes token to N_{j+1}
  end for
end while

Monadring employs several security measures to ensure fairness, unpredictability, and fault tolerance in this process. **Verifiable Random Functions (VRFs)** are used to determine the ring ordering or the initial token holder in each round, randomizing the token circulation sequence and making it difficult for adversaries to predict or influence whose turn comes next. This stochastic element prevents a malicious node from consistently positioning itself advantageously in the sequence. Additionally, Monadring can incorporate **Full Homomorphic Encryption (FHE)** techniques (or other cryptographic masking) to protect the contents of the token as it circulates. In essence, critical parts of the token (such as the state or certain event results) can be encrypted in a way that allows verification of correctness without revealing

raw data to intermediate nodes. This means a node holding the token cannot glean the outcomes of another node's execution for certain sensitive events, mitigating the risk of strategic behavior or information leakage in small networks. The combination of VRF-based randomness and FHE-induced confidentiality creates a game-theoretic equilibrium similar to a one-shot Prisoner's Dilemma: even with few participants, nodes are incentivized to behave honestly since they cannot easily predict or benefit from the next turn, nor can they see others' uncommitted results to copy or sabotage them.

To further secure the subnet, Monadring initializes a **threshold key-sharing** scheme among the participating validators. At the genesis of a subnet (or when the subnet membership changes), the nodes collaboratively generate a set of shared public/private keys such that a threshold (e.g. a majority) of them must cooperate to produce a valid signature. These **threshold signatures** are used for important collective actions like signing off the new subnet state or any external outcome that the subnet wants to present to the outside world (for example, posting a verified result to the main chain or controlling a joint account on another blockchain). The threshold cryptography[1989] ensures that no single node can unilaterally forge a state confirmation or external transaction—at least a quorum must agree, which aligns with the consensus. Monadring's design includes provisions for **key resharing** as well: if the subnet's validator set $\mathcal{S}_i$ changes (nodes leave or join), the threshold key can be regenerated or updated without exposing the secret, preserving security continuity as the subnet evolves.

Robust **fault tolerance** is built into Monadring to handle node failures or malicious actors. Since the token passing is sequential, a stalled or malicious node (one that refuses to pass the token or goes offline while holding it) could halt progress. To counter this, timeouts and bypass mechanisms are employed: if a node holds the token beyond a certain expected interval without producing a valid update, the protocol can trigger a timeout and effectively skip that node or revert to a safe last state, passing the token to the next node in line. In extreme cases, a misbehaving node can be flagged for removal from the subnet by the consensus of honest nodes. This ensures that the subnet continues to make progress (liveness) even in the presence of faults. In case an invalid state or event is detected (e.g., a node appends a malformed or unauthorized event), other validators will reject it during their validation step — the token can then be "rolled back" to the last consistent state, and the offending node's contribution ignored, maintaining consistency. Through these measures, Monadring achieves a reliable consensus within each subnet, even when the subnet might be as small as a few nodes operating in a partially trustless environment.

In summary, Monadring provides Verisense with an efficient subnet consensus layer that balances flexibility with security. By confining consensus to a rotating token and using cryptographic safeguards (VRFs for randomness, FHE for privacy, threshold signatures for collective trust, etc.), it enables even small-scale validator groups to reach agreement on state updates fairly and securely. The Monadring consensus instances run under the oversight of the main Verisense chain, which records each subnet's state roots and membership. This hierarchy means that while each subnet (each running Monadring) operates mostly independently for high performance and custom functionality, the outcomes are anchored to the main blockchain's security. The practical implication is that Verisense can host many **attested validation service (AVS)** subnets in parallel – each tailored to a specific application's needs – without sacrificing the overall trustworthiness of the network. These subnets can rapidly process application-specific events (including off-chain interactions) and then periodically commit succinct proofs or state digests to the main chain for verifiability.

3.2.2 Nucleus

A Nucleus in Verisense is the instantiation of the Monadring subnet concept for a particular decentralized application or agent. It represents a second-layer runtime environment where the application's logic is executed by a dedicated set of validators in a subnet. Each Nucleus is deployed as WebAssembly (**WASM**) bytecode, meaning the developer writes the application logic (e.g. in Rust or another supported language) and compiles it into a deterministic WASM module. This module is then installed on the Verisense Hostnet via a special transaction, which effectively **creates the Nucleus**. The Hostnet assigns a subset of validators (the chosen $\mathcal{S}_i$ for this Nucleus) to spin up the Nucleus instance. These validators collectively run the WASM code in parallel, each within a sandboxed VM, to maintain the Nucleus's state and perform its computations. The degree of decentralization for a Nucleus (i.e. the size of $\mathcal{S}_i$) is flexible: a Nucleus can be secured by a handful of nodes for lightweight, low-cost applications or by a larger quorum for higher-stakes applications that demand stronger security. This allows developers to **tune the security-cost tradeoff** by selecting how many validators will underpin their application's execution. Notably, Verisense uses a **"reverse gas" model for Nuclei** – end-users by default do not pay transaction fees to interact with a Nucleus. Instead, the Nucleus publisher (developer) funds the execution, similar to a cloud service model. This lowers the barrier for users to use these decentralized apps, making interactions feel like Web2 experiences (free reads and writes, unless the developer imposes restrictions or payments for specific operations).

The execution model of a Nucleus leverages the Monadring consensus to ensure that all validator nodes stay in sync through a consistent ordered log of events. From the perspective of the Nucleus's logic, events can be things like

function invocations (transactions), timers firing, or external data requests. These events are injected into the Monadring token rotation of that Nucleus's subnet. Thus, even though each validator runs the WASM code, only the **token-holding node** at any given time will perform certain actions, especially those that are nondeterministic or involve external systems, while the others will verify and replay them. For example, if the Nucleus needs to fetch data from an external API (an HTTP request), one node (the one with the token when the event is due) will actually perform the HTTP request. When it receives the response, it packages the result as a response-event in the token (often including cryptographic evidence like a TLS handshake proof or the server's certificate and a hash of the response). As the token circulates, all other validators see the request event (which they all would have initiated logically) and then the response event (carrying the actual data). Using shared keys (from a Diffie-Hellman handshake recorded deterministically) all nodes can **decrypt and verify the response** in exactly the same way, thereby reaching the same resulting state as the node that made the request. This two-step event design (request event and response event) ensures that external network calls, which are inherently unpredictable and asynchronous, do not break consensus — every node ends up with the same response data and state update despite only one node actually interacting with the outside world. A similar approach is taken for **timers** (scheduled events): when a timer is set in the Nucleus, it is logged as an event. When the time elapses, only one node will generate the "timer trigger" event (because its local clock alarm went off), but that trigger is then carried in the token to other nodes, which recognize it and thereby all agree the timer has fired and should be handled, disabling their own redundant timers. These mechanisms allow Nuclei to have rich, *active* behavior (like making web requests or scheduling tasks) while still maintaining deterministic consensus across the validator set.

Each Nucleus maintains its own state (for instance, a key-value store using RocksDB is provided for storage, isolated per Nucleus). After each round of the Monadring or after a certain number of events, the Nucleus's state is **check-pointed** and a state root (cryptographic hash of the state) is produced. Thanks to the threshold key established among the Nucleus's validators, the subnet can produce a **collective signature** on this state root (attesting that a quorum of the validators agree on the state). The Verisense Hostnet (main chain) receives these signed state roots and records them in its ledger. This design forms a robust accountability mechanism: even though the heavy computation and interactions happen off the main chain inside the Nucleus, the outcomes (in the form of state hashes and logs of events) are periodically anchored to the main chain, where they are immutable and publicly verifiable. If a dispute or verification is needed, one can refer to the main chain's record or even replay the Nucleus's logged events from a known good state root. In case of catastrophic failure or a need to **rollback** the Nucleus, the main chain's last recorded state root can serve as a secure point of reference to restart or restore consistency. Because the Nucleus's code is part of its state (the code itself is installed as the first event of the Nucleus), upgrades to the application logic are also handled as state updates – deploying a new version of the WASM code is just another event in the sequence, agreed upon via the same consensus process, which makes versioning and patching a controlled and transparent operation.

From a practical standpoint, Nucleus subnets empower developers to build complex decentralized applications with capabilities far beyond typical smart contracts. A Nucleus can interact with external web services (fetch data, call APIs), incorporate off-chain computation (e.g. running AI models or complex algorithms), and even control assets outside of Verisense. For instance, using the threshold signature capability established by Monadring, the validators of a Nucleus could jointly manage a private key for an external blockchain account (via a threshold signature scheme, TSS). In effect, the Nucleus could **act as a bridge or oracle**, signing transactions on another chain (Bitcoin[2008], Ethereum[2014], etc.) without any single node ever accessing the full private key. This opens the door to "bridgeless" cross-chain integration – the Nucleus itself can trustlessly initiate actions on external networks or handle tokens from other chains, governed by the consensus of its validators. Meanwhile, the rich **Verisense SDK** available to Nucleus developers provides high-level primitives (for networking, cryptography, scheduling, etc.), making it easier to build these advanced functionalities. All of this is accomplished with the assurance that the Nucleus's behavior is **verifiable**: every action is either agreed upon by multiple nodes or notarized on the main chain. Thus, the Nucleus serves as the execution engine of Verisense's active blockchain architecture, where the **theoretical benefits** of the Monadring consensus (scalability through subnets, off-chain activeness, enhanced privacy and security) directly translate into **practical applications**. Developers can create decentralized services that are interactive and high-performance, while users benefit from trust-minimized outcomes (thanks to the cryptographic commitments on the main chain and the multi-node validation).

In summary, the Nucleus component transforms the Verisense network into a platform for scalable multi-agent orchestration, by running each application in a controlled, verifiable mini-blockchain (the subnet) that seamlessly interfaces with both the Verisense main chain and the outside world.

Within each Monadring, decentralized execution environments called **Nuclei** are deployed. Each Nucleus serves as a secure runtime capable of proactively initiating and managing diverse tasks. Leveraging Active Blockchain technology, a Nucleus can autonomously execute various operations, as shown in Table 1.

Table 1: Nucleus Operations, Descriptions, and Security Guarantees

Type	Description	Security Guarantee
HTTP/HTTPS Requests and Inter-Agent Communication (MCP)	Securely interacting with external services or other agents, integrating off-chain data into on-chain contexts.	Proof-of-Data-IO (zkTLS)
Timer or Scheduled Tasks	Performing delayed or scheduled computations autonomously, without external triggers.	System Scheduler Module
Threshold Secret Sharing (TSS)	Providing cryptographic security through distributed key management and multi-party computation.	Shamir Secret Sharing
Large Language Models (LLMs) and Agent-to-Agent (A2A) protocols	Facilitating sophisticated semantic processing and communication between decentralized AI agents.	Proof-of-Data-IO, Proof-of-Semantic
Bridges/Oracles	Enabling secure cross-chain asset transfers and off-chain data integrations.	Eigenlayer [2023] Restaking

- **HTTP/HTTPS Requests and Inter-Agent Communication (MCP)**: Securely interacting with external services or other agents, thus integrating off-chain data into on-chain contexts.
- **Timer or Scheduled Tasks**: Performing delayed or scheduled computations autonomously, without external triggers.
- **Threshold Secret Sharing (TSS)**: Providing cryptographic security through distributed key management and multi-party computation.
- **Large Language Models (LLMs) and Agent-to-Agent (A2A) protocols**: Facilitating sophisticated semantic processing and communication between decentralized AI agents.
- **Bridges/Oracles**: Enabling secure cross-chain asset transfers and off-chain data integrations.

This hierarchical arrangement—validators elected by VRF forming Monadrings, and Monadrings hosting proactive Nuclei—creates a robust, scalable, and highly secure architecture. It supports decentralized autonomous agents (dAgents) that are both secure (via decentralized consensus mechanisms like Proof-of-Data-IO and Proof-of-Semantic) and autonomously responsive (via Active Blockchain technology), effectively bridging traditional web services, advanced AI, and blockchain consensus in a coherent, decentralized ecosystem.

4 Active Blockchain based Decentralized Agent Hosting and Orchestration

4.1 Centralized vs. Decentralized Agent Models

To ground the discussion, we compare traditional centralized agents with decentralized blockchain-based agents across key dimensions. A *centralized agent* is typically deployed on a single trusted server or cloud platform under one entity's control, whereas a *decentralized agent* runs on a distributed network of nodes (e.g., an active blockchain) with no single point of control. Table 2 summarizes the differences in execution model, trust model, security, scalability, and interaction pattern between these two paradigms.

- **Execution Model:** *Centralized:* The agent's logic executes in one place (a single server or data center), which makes it simpler to manage state and resources. *Decentralized:* The agent's computation is replicated or partitioned across multiple independent nodes, coordinated via consensus. Each action or state update is agreed upon by the network, providing redundancy and fault tolerance at the cost of higher coordination overhead.
- **Trust Model:** *Centralized:* Users must trust the hosting entity to run the agent correctly and not manipulate outcomes—authority is vested in a single operator, so the system is not trustless. *Decentralized:* No single operator is trusted; instead, trust is distributed among many participants. The network relies on cryptographic consensus, meaning the agent's state and actions are verified by multiple validators. (However, in a decentralized setting the agent must often prove its identity and correctness to others, imposing a higher burden of proof for trust.)

- **Security:** *Centralized:* While the platform can enforce security policies internally, it remains vulnerable to single-point failure. A breach of the central server can compromise the agent and all of its data or actions. *Decentralized:* There is no single point of failure; the system can tolerate some fraction of compromised nodes (Byzantine failures) without losing overall integrity. Any data tampering by one node is detected by others during consensus. On the other hand, new attack vectors emerge in decentralized systems (e.g., a colluding majority of nodes performing a 51% attack) that are absent in a centralized model.
- **Scalability:** *Centralized:* Scaling the agent's performance is straightforward up to hardware limits (vertical scaling or adding more servers), but ultimately bounded by the capacity of the central infrastructure and its bandwidth. *Decentralized:* The agent can potentially leverage the collective resources of an entire network of nodes, improving throughput via parallelism. Yet, if every node must process every task (as in many blockchain designs), overall throughput is limited by the slowest nodes and network communication. Advanced architectures (e.g., sharding or subnetworks) can improve throughput, but coordination overhead is inherently greater than in a centralized setup.
- **Interaction Pattern:** *Centralized:* The agent interacts with external services or other agents directly via standard APIs or messaging, with all calls routed through the central host. These interactions are fast and flexible, but their correctness is not independently verifiable by outsiders (one must trust the agent's reports of external data or events). *Decentralized:* Interactions (whether between agents or with external systems) occur through the blockchain or via decentralized oracles. External API calls require special handling so that multiple nodes can agree on the result (often using quorum attestations or cryptographic proofs). Inter-agent communication typically happens by posting on-chain messages/events that are visible to all participants. This yields transparency and verifiability for every interaction, at the cost of additional latency and complexity in obtaining consistent external information across the network.

The comparison between centralized and decentralized agent paradigms across key factors is presented in Table 2.

Aspect	**Centralized Agents**	**Decentralized Agents**
Execution Model	Runs on a single server or controlled infrastructure (one execution context).	Runs across multiple nodes in a distributed network; actions are replicated and validated via consensus.
Trust Model	Requires full trust in a central authority or operator to be honest and secure.	Trust is distributed: relies on a trustless consensus of many validators; no single party can corrupt the system.
Security	Single point of failure — if the central node is compromised, the agent is compromised.	No single point of failure — resilient to individual node compromise (Byzantine fault tolerant)[1999], though majority collusion is a new threat.
Scalability	Vertical or horizontal scaling by adding resources to the central server; simple but ultimately limited by one coordinator.	Horizontal scaling by adding more nodes; greater aggregate resources, but consensus overhead can bottleneck throughput.
Interaction Pattern	Direct calls to external APIs and messages between agents, which are fast but opaque (not automatically verifiable by third parties).	Interactions mediated by blockchain (on-chain calls or oracles for off-chain data) — transparent and auditable, but require consensus on external inputs.

Table 2: Comparison of centralized vs. decentralized agent paradigms across key factors.

4.2 Security Risks and Attack Scenarios for Decentralized Agents

While decentralization removes the single point of failure, it also introduces new security challenges for agent-based systems. Some of the main risks and potential attack scenarios in active blockchain environments are outlined below:

- **API Response Forgery:** An adversary may attempt to forge or tamper with the data an agent fetches from external APIs/services. For example, a malicious oracle or compromised agent node might supply a fake API response to the blockchain, causing the agent to act on false information. If such manipulated off-chain data is not detected, it can lead to wrong on-chain decisions and financial loss.

- **Malicious Decision-Making:** An agent's decision process itself could be turned against the system. A compromised (rogue) agent might deliberately output harmful or incorrect results, or an AI-driven agent could be manipulated via adversarial inputs into violating its policy. Such malicious or errant decisions can misallocate resources, violate security constraints, or otherwise harm the integrity of the workflow.
- **Consensus Exploitation:** Attackers might exploit the blockchain's consensus mechanism to subvert the agent. For instance, a group of colluding validators controlling a majority of stake or hash power could attempt to approve a fraudulent agent action or censor/block the agent's legitimate actions (a classic 51% attack scenario). By manipulating consensus (e.g., voting to accept forged data or ignore certain triggers), adversaries can undermine the correctness of the agent's outcomes.
- **Sybil and Collusion Attacks:** In an open network, a single adversary can spawn many pseudonymous nodes (Sybil attack[2002]) or bribe honest nodes to form a colluding cartel. Multiple malicious nodes could then feed identical false readings or collectively vote in favor of a malicious outcome, making it appear "consensus-backed." Without safeguards (like identity/stake requirements), Sybil or collusion attacks can distort the agent's view of reality and break the assumption of independent validators.
- **Denial-of-Service:** Adversaries can attempt to disrupt the agent or the network's ability to reach consensus by overwhelming it. For example, flooding the agent with an excessive number of triggers or expensive tasks, or spamming the network with transactions, could slow down or stall the agent's workflow. Such DoS attacks aim to exhaust nodes' resources or delay time-sensitive agent actions, undermining the agent's reliability.

Despite these challenges, active blockchain architectures are uniquely capable of hosting and orchestrating decentralized agents securely. They achieve this by embedding defense mechanisms directly into the consensus process. In particular, active blockchains employ **IO-based consensus** and **semantic-based consensus** layers to catch and mitigate malicious behavior. *Proof-of-Data-IO* ensures that any off-chain input an agent brings on-chain is validated by multiple independent parties or accompanied by cryptographic evidence, thwarting API response forgery by requiring network-agreed verification of external data. Likewise, *Proof-of-Semantic* checks that an agent's actions and outputs semantically align with its expected goals or protocols (verifying intent and context), helping to detect nonsensical or malicious decisions before they affect the system.

Additionally, the underlying consensus protocol is designed to be Byzantine fault tolerant and is often augmented with economic penalties for malicious actors (e.g., slashing of staked tokens for dishonest validators). This deters collusion and Sybil attacks by making them costly or easily identified. All agent-triggered state changes must pass through these layered checks: any single node or minority coalition attempting an exploit will be outvoted or ignored by the honest majority. In essence, the active blockchain turns potentially vulnerable agent operations into verifiable transactions. Through cryptographic proofs and multi-party agreement, the network can reject forged inputs, flag abnormal agent outputs, and maintain liveness even under attack. This multi-tier security approach enables active blockchains to securely host and orchestrate decentralized agents despite adversarial conditions, providing a level of trust and reliability that is difficult to achieve in conventional decentralized systems.

4.3 Proof of Data-IO: Verifiable Off-Chain Interactions

In decentralized agent execution (e.g., blockchain or AI agents), having **verifiable off-chain data** is crucial. Agents often need to fetch external HTTPS data (API queries, web pages, etc.) as inputs, but without special measures, other parties must *trust* that the agent is providing genuine data. Traditional TLS ensures confidentiality and integrity between a client and server, but offers no way for a third party to verify the content of an HTTPS response [2023]. This gap motivates *Proof-of-Data-IO*: a cryptographic proof that an agent's off-chain input/output (I/O) interactions (such as web API calls) are authentic and untampered. By obtaining verifiable off-chain data, decentralized agents can prove the origin and integrity of their inputs, enabling trustless automation and on-chain use of real-world information without relying on centralized oracles. **zkTLS and Trustless Data Proofs:** *zkTLS* (zero-knowledge TLS) refers to a class of protocols that augment TLS connections with cryptographic proofs, so a client (prover) can convince a verifier about data obtained over TLS **without** revealing sensitive parts of the data itself. In essence, zkTLS protocols introduce a third party or proof system into the TLS handshake in a way that preserves end-to-end security yet produces evidence of the session's contents. This is essential for trustless Proof-of-Data-IO: it allows an agent to fetch HTTPS data and produce a **proof of authenticity** (and optionally of certain properties of that data) that anyone can verify. Using zkTLS, the agent's data input can be treated as provably correct, much like a blockchain transaction, eliminating blind trust. The verifier (which could be a smart contract or another user) can check that the data indeed came from a specific server and was communicated over a valid TLS session, all without needing to see any secret information (like API keys or the full plaintext). This capability enables **decentralized web oracles** and verified off-chain interactions, bridging Web2 data into trust-minimized Web3 systems.

Major zkTLS Protocols: Over the past few years, several protocols have emerged to realize verifiable TLS sessions and off-chain data proofs. Below we briefly overview the major approaches and their characteristics:

- **DECO:** *Decentralized Oracle for TLS*[2020] introduced the first privacy-preserving TLS oracle for modern TLS 1.2 without requiring any server-side modifications. In DECO, a three-party handshake is conducted between the client, server, and an "oracle" node. Most of the client's TLS handshake and decryption is carried out via a malicious-secure two-party computation (2PC) between the client and oracle. The oracle helps the client establish the TLS session and validate the data, but learns nothing about the plaintext. After retrieving data, DECO uses zero-knowledge proofs to allow the client to prove statements about the TLS session data (e.g. *"my account balance is above X"*) to a verifier, without revealing the raw data. DECO thus achieves strong privacy (selective disclosure of data) and works with unmodified servers. The trade-off is the use of heavy cryptography (general garbled circuits for 2PC and ZK proofs), which can be computationally intensive for large data.

- **zkPoD:** *Zero-knowledge Proof of Delivery*[2019] is a decentralized protocol for fair data exchange between untrusted parties. While not a TLS oracle per se, zkPoD tackles a related problem: ensuring that a data seller and buyer can engage in "payment-versus-delivery" of data without trusting each other or a third party. It uses the blockchain as an arbiter and employs zero-knowledge proofs to ensure *fairness* and privacy. For example, a seller can prove on-chain that the data they deliver satisfies certain properties (like containing certain records and no others) without revealing the data publicly, and the buyer's payment is locked until the proof and delivery are satisfactory. zkPoD focuses on selective queries over data (returning matching records) and hides both the data and query details from the blockchain. It requires no trusted hardware or third-party oracle, instead, cryptographic protocols (zero-knowledge Succinct Non-interactive ARguments of Knowledge, commitments, etc.) ensure that if either party cheats, the other can prove it. The outcome is a trustless "proof-of-delivery" that the buyer can use to verify they got exactly the promised data or get refunded – a different use-case from web oracles, but complementary in enabling verifiable off-chain data exchange.

- **Pado (Primus):** Pado Labs[2025] proposed an improved zkTLS protocol that introduces a *"garble-then-prove"* technique. Like DECO, it uses secure 2PC to emulate a TLS handshake without server cooperation, but Pado aims to reduce the heavy cost of generic MPC. Instead of performing a fully malicious-secure garbled circuit evaluation for every step of TLS, Pado executes the critical parts in a garbled circuit once and then uses more efficient proof techniques to attest to the correctness of the remaining steps. This approach dramatically reduces communication and computation: their implementation reports a 14× reduction in communication and about an order of magnitude less computation compared to DECO. Pado's protocol can be extended to TLS 1.3, addressing the fact that earlier schemes focused on TLS 1.2. In summary, Pado preserves *selective disclosure* and server transparency like DECO, but with better performance by minimizing the use of expensive MPC in favor of lighter cryptographic proofs.

- **TLS-N:** *TLS-N (TLS Non-repudiation)*[2018] is a research prototype that takes a different approach: it modifies the TLS protocol itself to embed proof-generation capabilities. TLS-N is a TLS *extension* (compatible with TLS 1.3) that allows the server and client to produce a **non-interactive proof** of the TLS session's contents, which can be efficiently verified by third parties or even smart contracts. In a TLS-N handshake, the server contributes additional cryptographic material (such as signatures or MACs over the transcript) so that, after the session, the client can output a proof package that anyone can validate without contacting the server. Importantly, TLS-N's proofs support *privacy*: the client can hide or redact parts of the TLS data (e.g. cookies, passwords) before generating the proof, so only the chosen content is revealed and verifiable. Because the proof is non-interactive and succinct, it's suitable for on-chain verification or offline verification by multiple parties. The downside is **server compatibility**: TLS-N requires servers to adopt the new TLS extension (and possibly obtain a new type of certificate), which is a non-trivial change to the Web ecosystem. Thus, TLS-N is a forward-looking solution for *"built-in"* web proofs, but not usable with legacy servers unless they update their TLS stack.

- **TLSNotary:** TLSNotary[2024] is one of the earliest and most influential protocols for verifiable TLS and has recently been redesigned (2022) with modern security and features. Like DECO and Pado, it works with unmodified servers by introducing an external verifier into the TLS connection via a three-party handshake. The prover (TLS client) and verifier perform a joint TLS session with the server using MPC, so that the verifier can attest to the data's authenticity while the server sees nothing unusual. The verifier is assured that the revealed parts are exactly as received from the server, thanks to the protocol's cryptographic guarantees. We dive deeper into TLSNotary below, as it provides a foundation for trustless Proof-of-Data-IO in decentralized systems.

4.3.1 TLSNotary Protocol

TLSNotary enables a prover and verifier to jointly establish a TLS connection and produce a transferable proof of the data exchanged. The protocol can be understood in three main phases:

- **MPC-TLS Handshake (Phase 1):** An MPC-assisted TLS handshake and data retrieval.
- **Notarization (Phase 1.5):** Notarization (signature) of the session data.
- **Selective Disclosure (Phase 2):** Selective disclosure to a verifier (followed by verification).

1. **MPC-TLS Handshake (Phase 1):** Instead of the client alone performing the TLS handshake with the server, the prover (client) and verifier run it *together* using secure two-party computation. The goal is to let the verifier validate the TLS **integrity** (that the handshake and record encryption are done correctly) without learning the plaintext traffic. To achieve this, TLSNotary employs a variant of *dual execution* MPC with enhanced privacy, dubbed **DEAP – Dual Execution with Asymmetric Privacy**.

   In practice, the prover and verifier each hold random secret shares of the TLS secrets and cooperate to compute the TLS protocol. For example, if the TLS handshake uses an ephemeral Diffie–Hellman key exchange, the client's ephemeral private key a can be split into two additively shared values a_P (prover's share) and a_V (verifier's share). The client sends the public value g^a to the server as usual (with $a = a_P + a_V$). The server responds with its public value g^b. Now the **Pre-Master Secret (PMS)** is the Diffie–Hellman value g^{ab}. The prover and verifier can each compute a component of this without revealing a or b to each other: the prover computes $X = (g^b)^{a_P}$ and the verifier computes $Y = (g^b)^{a_V}$. These are **multiplicative shares** of the PMS, since $X \cdot Y = (g^b)^{a_P} \cdot (g^b)^{a_V} = g^{a_P b + a_V b} = g^{ab} =$ PMS. They then run a small sub-protocol to convert this into additive shares of the same PMS (this is the *M2A: multiplicative-to-additive* share transformation).

   Throughout the handshake, similar techniques are used: using efficient MPC sub-protocols, the two parties jointly compute the TLS key schedule and encryption/decryption of data. For operations like AES-GCM encryption and the GHASH authenticity check, TLSNotary uses optimized conversions between arithmetic and boolean shares (**A2M/M2A** transformations as introduced by Yu *et al.*) and garbled circuits for any complex checks. Notably, DEAP (the dual execution approach) means the prover and verifier effectively execute the protocol twice in a asymmetrically private way: by swapping roles in a second execution or by having one party reveal certain inputs at the end, they can detect any cheating without leaking the actual data. A lightweight **garbled circuit** equality check is performed on the final results to ensure both parties obtained the same TLS handshake outcomes (e.g. the same session keys or verify data) and neither diverged from the protocol. If this check passes, the verifier is assured that the TLS session with the server was established correctly and that the server's data received by the client is authentic (came from the legitimate server and passes all TLS integrity checks), all without the verifier seeing the plaintext of that data.

2. **Notarization (Phase 1.5):** Once the TLS session is complete and the data is obtained by the prover (and its authenticity is validated by the verifier through the MPC process), TLSNotary can produce a **portable proof** of this interaction. In the updated TLSNotary architecture, this is achieved by introducing a *general-purpose* verifier called a **Notary**. The Notary's role is to **cryptographically sign** a commitment to the TLS session data (and relevant metadata like the server's identity). Essentially, the Notary acts as an impartial witness to the session: it participated in the handshake (Phase 1) and is convinced of the data's authenticity, so it now "notarizes" the result.

   For example, let D denote the full data retrieved from the server (e.g. an HTTP response) and let $H(D\|\text{server\_id})$ be a hash of the data and server's identity. The Notary generates a digital signature $\sigma_N = \text{Sign}_{sk_N}(H(D \parallel \text{server\_id}))$ attesting that *"this data came from the server with identity server_id in a valid TLS session"*. The pair (D, σ_N) (or any selective subset of D, see below) is a **Proof-of-Data-IO** that can be later presented to any third party. This notarization step makes the proof *reusable*: the prover can now go offline, and at any future time show the signed data to another verifier (or even to a smart contract) as evidence, without needing the Notary to be involved again.

   We emphasize that TLSNotary achieves this without any server cooperation – the server is unaware that the data it sent has been "notarized," and importantly the server cannot prevent it or watermark the data to detect it. The Notary's signature vouches for the data's authenticity, so as long as verifiers trust the Notary's public key (or a set of Notaries, to mitigate trust in any single one), the proof is trust-minimized. This concept is powerful in decentralized systems: for instance, a Notary signature can serve as a cross-chain credential – an Ethereum smart contract or a Polkadot[2016] parachain could accept σ_N as proof that D is valid, provided the Notary's key is recognized in their trust setup.

3. **Selective Disclosure & Verification (Phase 2):** With the notarized transcript in hand, the prover has fine-grained control over what to share with an end verifier. They may choose to reveal the entire data D, or only

specific parts of it, depending on the use case. TLSNotary supports **selective disclosure** natively. For example, if D is a bank statement page, the prover might disclose only the line showing their account balance and redact other transactions. Along with the disclosed portion $d \subset D$, the prover provides the Notary's signature σ_N and any additional cryptographic evidence needed to tie d to the signed $H(D)$. This might be done by revealing a Merkle proof or offsets and hashes for the redacted portions so that the verifier can independently compute $H(D)$ and check σ_N. Because the Notary signed the *commitment* to the full data, as long as the verifier can validate that the shown piece d indeed came from some data D that hashes to the signed value, the verifier is convinced of d's authenticity.

The verifier then checks σ_N (using the Notary's public key) and the server's certificate to ensure the server identity in the commitment is the expected one, via the usual PKI trust chain. If all checks out, the verifier accepts the proof. Crucially, the verifier learns nothing about the redacted parts of D. The prover could even go further and prove **in zero-knowledge** a predicate about the data: for instance, instead of revealing an exact balance, they could provide a ZK proof that the hidden balance in D exceeds a threshold, or that a date of birth in D implies age ≥ 21. This is an advanced selective disclosure where the verifier only learns the truth of a statement (age is over 21) but not the sensitive data itself (the actual birthdate). Thus, TLSNotary allows *minimal disclosure*: the prover shares only what is necessary for the task, preserving privacy.

Advantages in Decentralized Systems: TLSNotary's approach has several key benefits for integrating off-chain data into blockchain and decentralized applications. First, it requires no change or permission from existing Web2 servers, yet can produce proofs about their data – any HTTPS API or website can be used as a data source. This greatly broadens the scope of data available to smart contracts and trustless agents, compared to schemes that require server support or special oracles. Second, the proofs are **portable and reusable**: once the Notary signs the data, the same proof can be presented to multiple different verifiers or even across different blockchains.

For example, a user could obtain a notarized proof of their credit score from a web service and then reuse that single proof to apply for loans on several DeFi platforms. This reuse is far more efficient than querying the web service separately for each platform. It also enables **cross-chain data portability** – a proof generated off-chain can be verified on any chain (provided the Notary's signature or a succinct proof is recognized), avoiding siloed oracles per chain. Third, the scheme is **privacy-preserving** by design, which is critical when bringing personal or proprietary data on-chain. Users retain full control over what they reveal. This mitigates one of the biggest hurdles for decentralized identity and credit systems: proving facts from Web2 platforms (like KYC info, social media reputation, bank balances) without doxxing oneself. Finally, TLSNotary offers a path to decentralize the role of the Notary itself. In practice, a network of independent Notaries could exist, and a proof could require a quorum signature or multi-party attestation, reducing the need to trust any single verifier. Ongoing research is also improving efficiency: for instance, replacing parts of the MPC with succinct zero-knowledge proofs (such as *VOLE-based interactive ZK* protocols) could make the client's proof output even more compact and directly verifiable on-chain.

In summary, **Proof of Data-IO** via TLSNotary (and related zkTLS protocols) empowers decentralized agents to interact with Web2 APIs in a trustless manner. It combines the best of both worlds: the rich data of the Internet and the strong integrity guarantees of cryptography and blockchains. By leveraging zkTLS techniques like TLSNotary, we can ensure that *"off-chain"* does not mean *"off-the-record"* – every critical off-chain interaction can be accompanied by a cryptographic proof, anchoring real-world data securely into our decentralized systems.

4.3.2 Adaptation of TLSNotary Protocol for Verisense

Verisense's Proof-of-Data-IO mechanism is heavily inspired by the TLSNotary protocol, but includes several architectural and implementation-level adaptations to fit the Verisense framework. Specifically, integrating with the Verisense Nucleus execution environment required developing a modular, programmable interface to launch and verify TLSNotary sessions as autonomous agent tasks. The design also features enhancements for compatibility with Verisense's active scheduling model (e.g., timed data fetches and reactive calls), ensuring that proofs can be generated and verified within scheduled or event-driven workflows. Protocol-level adjustments were made to improve interoperability with on-chain smart contracts, including flattening the proof output format and embedding dedicated smart contract verifiability tags in the proofs. Importantly, while the core ideas from TLSNotary are preserved (e.g., the MPC-TLS handshake and selective disclosure), Verisense generalizes and modularizes these components to support decentralized agent orchestration at scale.

4.4 Proof of Semantic

Ensuring semantic consistency among decentralized large language model (LLM) outputs is a non-trivial challenge due to the inherent variability and probabilistic nature of text generation. To address this, we propose a novel *Proof-of-*

Semantic protocol within the Verisense network. This protocol cryptographically guarantees that the selected output among decentralized agents aligns most closely with a specified semantic intent or query. Crucially, it maintains the confidentiality of non-selected outputs throughout the evaluation process.

The protocol consists of three distinct phases:

(1) Decentralized Embedding and Encryption. Each agent i computes an embedding vector $E_i = f(O_i)$ of its generated output O_i, where $f(\cdot)$ represents a semantic embedding function (e.g., Sentence-BERT [2019]), mapping textual outputs into a semantic vector space $\mathbb{R}^d$. To ensure privacy and facilitate secure collective processing, agents encrypt their embeddings using a threshold fully homomorphic encryption (FHE) scheme, specifically leveraging the CKKS scheme [2017] for its efficient handling of real-valued vector arithmetic. Each agent computes an encrypted embedding $C_i = \mathsf{Enc}_{pk}(E_i)$ using the public key pk shared by the Verisense network. The corresponding secret key is distributed in a t-of-n threshold scheme among network nodes, ensuring no single node can independently decrypt the embeddings.

(2) Homomorphic Distance Computation and Semantic Consensus. Upon collection of encrypted embeddings $\{C_i\}$, the Verisense nodes collaboratively perform semantic similarity computations in ciphertext domain. Given a topic embedding $T \in \mathbb{R}^d$, representing the intended semantic target (derived from a query or predefined domain embedding), the network computes encrypted squared Euclidean distances homomorphically:

$$D_i^2 = \sum_{j=1}^{d} (E_{i,j} - T_j)^2.$$

Leveraging CKKS's SIMD (Single Instruction Multiple Data) capabilities, this operation efficiently yields encrypted distances $C_i^D = \mathsf{Enc}_{pk}(D_i^2)$ for each agent. Nodes then determine the index $k = \arg\min_i D_i^2$ securely, employing privacy-preserving comparison techniques (e.g., garbled circuits or partial threshold decryption). To resolve cases with multiple semantically equivalent candidates (distances within a small threshold ϵ), a probabilistic tie-breaking mechanism selects among them, ensuring fairness and decentralization without deterministic bias.

Critically, the consensus selection occurs entirely in the encrypted domain, thus preserving the confidentiality of individual agent outputs and preventing inter-agent influence.

(3) Threshold Decryption and Final Output Retrieval. After identifying the winning agent k, the network jointly decrypts the corresponding ciphertext C_k using threshold decryption. Nodes contribute partial decryptions using their respective secret key shares, collectively reconstructing the embedding E_k. The actual textual output O_k is retrieved via a cryptographic commitment or supplementary encryption provided initially by the agent. Hence, only the selected agent's output is revealed, and all other outputs remain encrypted indefinitely.

The described protocol ensures that the final selected output is cryptographically verified as semantically closest to the intended query or topic. Algorithm 2 formalizes the *Proof-of-Semantic* process.

This protocol integrates state-of-the-art techniques from homomorphic encryption [2017; 2020] and semantic embedding methods [2019], establishing a robust and privacy-preserving mechanism for decentralized semantic consensus among AI agents. By enforcing semantic coherence cryptographically, the *Proof-of-Semantic* protocol significantly enhances the reliability and integrity of decentralized LLM-based systems.

To advance the practical application of Fully Homomorphic Encryption (FHE) within decentralized semantic consensus protocols, we are establishing a collaborative partnership with Sunscreen Labs[2025]. By integrating Sunscreen's expertise in FHE tooling with our semantic consensus framework, we aim to explore and develop scalable, privacy-preserving solutions that enhance the security and integrity of decentralized systems. This collaboration is expected to yield significant advancements in the deployment of FHE technologies, contributing to the broader adoption of privacy-centric computing paradigms.

Example Scenario: Quantitative Trading Agent Consensus Consider a practical scenario involving a decentralized network of quantitative trading agents aiming to determine the optimal entry position for a particular stock. A human operator or automated strategy specifies a query such as: *"What is the optimal price range to initiate a long position on stock XYZ today?"*

Each decentralized agent independently generates an analysis, producing textual outputs such as:

- Agent A: "Enter a long position between \$100 and \$105."
- Agent B: "Optimal buy zone is \$98 to \$102."

Algorithm 2 Verisense Semantic Consensus Protocol (Proof-of-Semantic)

Require: Query/topic Q, semantic embedding $T = f(Q)$, agent outputs $O_1, \ldots, O_n$
Ensure: Output O_k best matching the semantic intent

for agent $i = 1$ to n **do**
  $E_i \leftarrow f(O_i)$ // Embedding computation (e.g., SBERT)
  $C_i \leftarrow \mathsf{Enc}_{pk}(E_i)$ // Threshold FHE encryption (CKKS)
end for
Submit all C_i to Verisense network
for each ciphertext C_i **do**
  Compute $C_i^D \leftarrow \mathsf{Enc}_{pk}\left(\sum_j (E_{i,j} - T_j)^2\right)$ // Homomorphic distance
end for
Determine $k = \arg\min_i D_i^2$ securely
/* Optional probabilistic tie-breaking if multiple distances within threshold ϵ */
$\widetilde{C} \leftarrow C_k$ // Select winning ciphertext
$E_k \leftarrow \mathsf{ThreshDec}(\widetilde{C})$ // Threshold decryption
Retrieve original output O_k // Via cryptographic commitment
return O_k

- Agent C: "Initiate buys around \$103 to \$107 for best risk-reward."
- Agent D: "Recommend entry at \$99 to \$101."

Each agent computes an embedding of its output using a semantic embedding model such as Sentence-BERT. These embeddings are encrypted individually using threshold FHE (CKKS) to ensure privacy. The network then collectively computes the homomorphic distances between each encrypted embedding and the embedding of the query itself, reflecting the semantic closeness of each agent's recommendation to the intended trading query.

The semantic consensus step identifies the encrypted embedding closest to the original query embedding without revealing the details of other embeddings. Suppose the encrypted embeddings yield the following semantic distances (hidden under encryption initially):

$$D_A^2 = 0.08$$
$$D_B^2 = 0.03$$
$$D_C^2 = 0.11$$
$$D_D^2 = 0.05$$

Homomorphic computations and privacy-preserving comparison reveal that Agent B's embedding has the minimal semantic distance. After the network securely identifies Agent B's embedding as the closest, threshold decryption is performed exclusively on Agent B's encrypted output. The plaintext recommendation "Optimal buy zone is \$98 to \$102" is then retrieved and presented as the final network consensus.

This cryptographically ensures the semantic coherence of the chosen recommendation, protects the confidentiality of the other agents' outputs, and provides a robust mechanism for trustless semantic alignment in decentralized quantitative trading scenarios.

4.5 Embedding Fingerprint

Modern decentralized AI networks require mechanisms for clearly establishing ownership and accountability of individual AI agents' contributions, particularly in consensus-driven settings. Inspired by Sentient's OML[2024] fingerprinting mechanism, we introduce an AI-native cryptographic fingerprinting approach implemented directly within the encryption layer, rather than embedding fingerprints in the model parameters themselves. Instead of planting backdoor triggers in the model (as done in Sentient's fingerprinting scheme), we embed a verifiable *fingerprint* directly into the CKKS homomorphic encryption noise of each model's output ciphertext. This means that each agent's output is cryptographically "watermarked" without modifying the model's architecture or parameters. The key intuition is that the CKKS scheme's approximate encoding leaves room to imprint identifying information in the least significant bits of the encrypted message, which can be leveraged for origin verification. By manipulating the encryption noise (the small error term inherent in CKKS encoding) to carry a secret code, an agent can later prove ownership of an inference result. Crucially, this approach preserves the model's utility (the fingerprint perturbation is negligible for the model's accuracy)

Algorithm 3 Embedding Fingerprint and ZK-Proof Verification

Require: Agent embedding output $\mathbf{v} \in \mathbb{R}^d$, fingerprint identifier $\mu_i \in \{0,1\}^\ell$, public key pk, noise scale factor $\eta \ll 1$
Ensure: Fingerprinted ciphertext C_i, zero-knowledge proof Π_i

$\boldsymbol{\delta}_i \leftarrow \mathsf{EncodeFingerprint}(\mu_i, \eta)$ // Small perturbation embedding fingerprint
$\tilde{\mathbf{v}}_i \leftarrow \mathbf{v} + \boldsymbol{\delta}_i$ // Fingerprint-embedded embedding
$m_i(X) \leftarrow \mathsf{EncodeCKKS}(\tilde{\mathbf{v}}_i)$ // Encode to plaintext polynomial
$C_i \leftarrow \mathsf{Enc}_{pk}(m_i(X))$ // CKKS encryption embedding fingerprint in noise
$\Pi_i \leftarrow \mathsf{ZKProve}\left(\exists\, \tilde{\mathbf{v}}_i, r_i : C_i = \mathsf{Enc}_{pk}(m_i(X); r_i) \wedge \mathsf{HasFingerprint}(\tilde{\mathbf{v}}_i, \mu_i)\right)$ // Generate ZK-Proof
Submit (C_i, Π_i) to verifier/network
Verification:
  $b \leftarrow \mathsf{ZKVerify}(\Pi_i, C_i, pk, \mu_i)$ // ZK-Proof Verification
if $b = 1$ **then**
    Accept (C_i, Π_i) // Fingerprint verified
else
    Reject (C_i, Π_i) // Invalid fingerprint proof
end if
return (C_i, Π_i) if accepted, otherwise abort

and is entirely model-agnostic—any model's output can be fingerprinted at encryption time without additional training or architectural changes.

Formally, let $\mathbf{v}$ be the plaintext embedding output produced by a model. Each agent i has a unique fingerprint identifier (e.g., a binary string or integer) denoted μ_i. We define a small perturbation vector $\boldsymbol{\delta}_i$ encoding this identifier, such that $\boldsymbol{\delta}_i$ is sparse or of very low magnitude relative to $\mathbf{v}$. For example, $\boldsymbol{\delta}_i$ could embed the bits of μ_i into the k least significant bits of $\mathbf{v}$'s numerical representation. The fingerprinted plaintext is then $\tilde{\mathbf{v}} = \mathbf{v} + \boldsymbol{\delta}_i$. We ensure $\|\boldsymbol{\delta}_i\|$ is so small that $\tilde{\mathbf{v}}$ is perceptually and functionally indistinguishable from $\mathbf{v}$ (preserving the model's utility), yet $\boldsymbol{\delta}_i$ will be preserved through encryption and decryption as part of the noise term. When $\tilde{\mathbf{v}}$ is encrypted with CKKS to yield ciphertext $C_i = \mathsf{Enc}_{pk}(\tilde{\mathbf{v}})$ under the public key pk, the fingerprint μ_i becomes embedded in the ciphertext's noise. In practice, CKKS encryption of a real vector involves scaling by a large Δ and rounding; by carefully choosing $\tilde{\mathbf{v}}$, the rounding "error" encodes the fingerprint. One can think of this as defining a specific low-order bit-pattern in the plaintext polynomial $m(X)$ such that $m(X) = \mathsf{encode}(\mathbf{v}) + F_i(X)$, where $F_i(X)$ represents the fingerprint polynomial carrying μ_i. The encryption C_i thus carries $F_i(X)$ hidden in its least significant coefficients. This construction does not rely on any specific model architecture or weights—only on the encryption process—making it highly modular.

The verification of such an embedded fingerprint is achieved via a *zero-knowledge proof (ZK-proof)* on the ciphertext. After producing C_i, the agent also generates a ZK-proof Π_i attesting to the statement: "There exists a plaintext $\tilde{\mathbf{v}}$ such that $C_i = \mathsf{Enc}_{pk}(\tilde{\mathbf{v}})$ and $\tilde{\mathbf{v}}$ contains the fingerprint μ_i in the predefined format." Using modern zero-knowledge proof systems, the agent can prove knowledge of the encryption's plaintext and fingerprint without revealing $\tilde{\mathbf{v}}$ itself. The proof Π_i essentially certifies that the agent's claimed fingerprint is indeed embedded in the ciphertext's noise. Verifiers (which could be other agents, a smart contract, or the protocol's consensus nodes) check Π_i to confirm the presence of μ_i in C_i. If the proof verifies, the ciphertext is accepted as fingerprinted by agent i; otherwise it is rejected as non-compliant. This ZK-proof mechanism ensures that fingerprint verification does not require decrypting the content or exposing the fingerprint to others: only the validity of the statement is revealed, nothing more.

The Algorithm 3 cleanly separates the concerns of model performance and ownership verification. Unlike the Sentient OML approach, which requires training the model with special key-response pairs (and potentially modifying its behavior), this encryption-based fingerprint does not interfere with the model's internals at all. It thus supports a **model-agnostic deployment**: any model can be wrapped in this fingerprinting mechanism on-the-fly. Moreover, because the fingerprint is applied at encryption time, it composes naturally with **threshold FHE** setups. Even if decryption is distributed among multiple parties (threshold decryption), the fingerprint remains intact in the ciphertext until the final step. Verification via ZK-proof can be performed before any partial or threshold decryption occurs, ensuring that only ciphertexts with valid fingerprints will be chosen for decryption. This means the protocol can enforce that all participating agents embed their fingerprints, and only a ciphertext proven to carry a legitimate fingerprint is ever decrypted, adding an extra layer of trust in collaborative settings.

Another benefit of this approach is its compatibility with existing fingerprinting frameworks like Sentient's OML. Both backdoor-based fingerprints (embedded in the model) and encryption-based fingerprints (embedded in the ciphertext) could co-exist to provide defense in depth. For instance, a model owner could OMLize their model with robust backdoor triggers for later auditing *and* require that any use of the model in a protocol like ours involves encryption fingerprinting

for immediate, on-chain verification. The two techniques operate at different layers (model vs. encryption) and reinforce each other: the OML backdoors allow offline or ex-post facto ownership claims and detection of misuse, while the CKKS noise fingerprints enable real-time verification of output authenticity during protocol execution. This co-existence can make the system more resilient— even if an adversary somehow removed or bypassed the model's internal fingerprints via fine-tuning or distillation, they would still face the cryptographic fingerprint checks at usage time. In summary, embedding fingerprints in homomorphic ciphertexts offers a highly modular and secure way to prove semantic contribution or ownership, complementing the optimistic security of OML with a cryptographic guarantee that is immediate and does not rely on secret triggers in the model.

4.6 Hosting Agent Framework: Nuclues Agent

In the Verisense architecture, the Nucleus provides a secure and decentralized execution environment, enabling users to construct and deploy autonomous decentralized agents (dAgents) onto the Verisense network. The hosting and orchestration of these agents are driven by the proactive invocation capabilities provided by the Active Blockchain, while their security is guaranteed through two critical consensus algorithms, as illustrated in Fig 3.

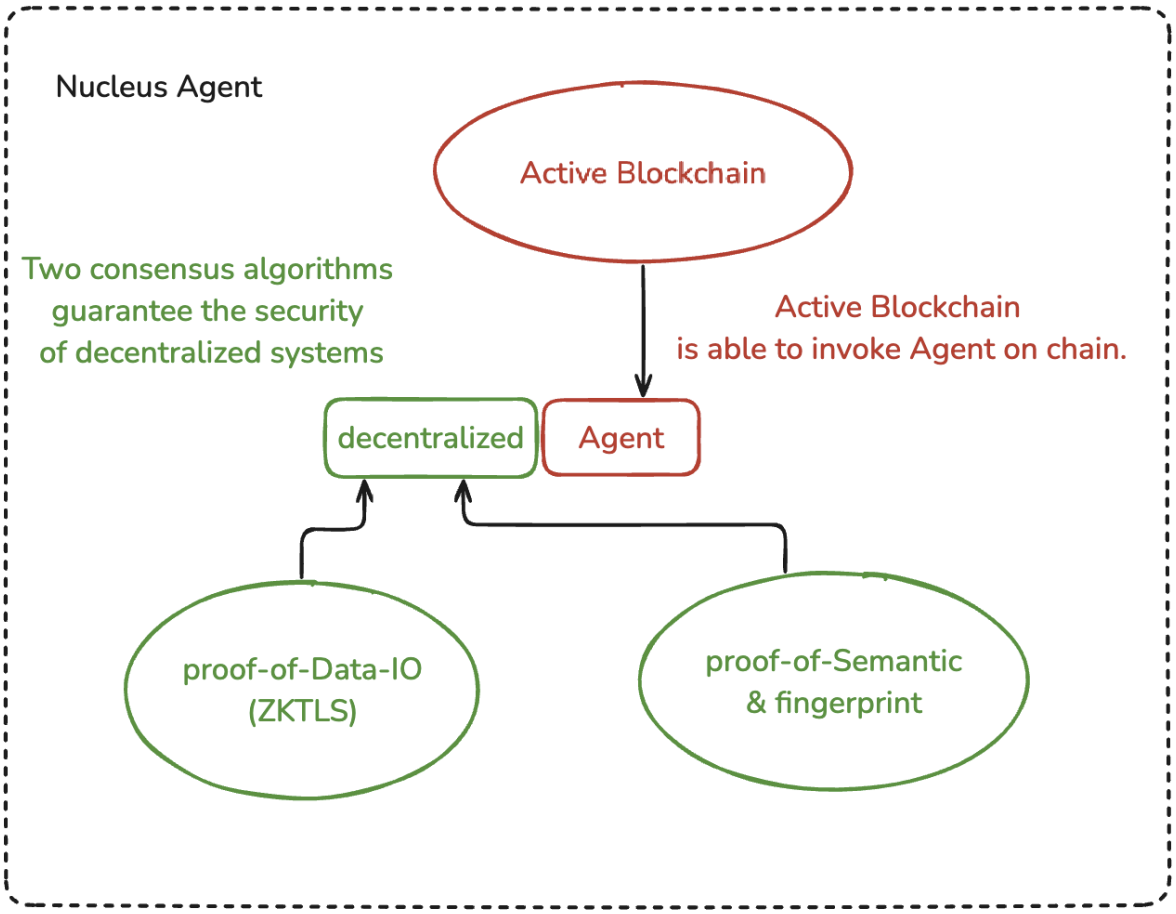

Figure 3: Illustration of Nucleus Agent Framework

As illustrated in the figure, the architecture for decentralized Agents (dAgents) within the Verisense network is jointly enabled by two crucial components: the Active Blockchain and two decentralized consensus mechanisms—**Proof-of-Data-IO (zkTLS)** and **Proof-of-Semantic**.

- The Active Blockchain component proactively handles the invocation and orchestration of the Agent logic. Unlike traditional passive blockchain architectures, Active Blockchain actively triggers on-chain agents,

enabling them to autonomously respond to external and internal events without waiting passively for user transactions or external interactions.

- The decentralized aspect and security guarantees of dAgents are ensured by two consensus algorithms:
  - Proof-of-Data-IO (zkTLS) verifies and guarantees the authenticity of external data interactions, ensuring trustless off-chain data integration.
  - Proof-of-Semantic cryptographically ensures that agent-generated outputs are semantically aligned with specified intents, preventing malicious or irrelevant responses from influencing the system.

The Nucleus environment facilitates building secure, autonomous Agents deployed directly onto the Verisense network, leveraging both proactive blockchain invocation and decentralized consensus security. Thus, through the complementary roles of proactive invocation by Active Blockchain and decentralized verification via dual-consensus algorithms, the robust, secure, and truly decentralized Agent orchestration paradigm is fully realized.

5 VeriFlow: Decentralized Agent Orchestration System

Building upon the Active Blockchain paradigm and the previously introduced *Proof-of-Data-IO* and *Proof-of-Semantic* mechanisms, this paper introduces **VeriFlow**, a decentralized orchestration framework designed for trustless workflow execution. VeriFlow empowers users to construct sophisticated workflows consisting of multiple decentralized **blocks** or **nodes**, analogous to visual pipelines offered by platforms such as Dify [2025] or DagWorks[2023].

Unlike traditional orchestration tools, VeriFlow workflows are compiled and executed directly on the blockchain-based Verisense Network, ensuring transparency, immutability, and verifiable execution at every step. This decentralized structure allows multi-step agent orchestrations without reliance on centralized coordinators or off-chain servers, thereby eliminating single points of failure and enhancing trust through on-chain enforcement of logic.

In a VeriFlow workflow, each block can securely invoke decentralized large language models (LLMs) from `0G`[2024], Ambient[2024] or Bittensor[2020], or autonomous agents built with agent framework `ElizaOS`[2025]. VeriFlow utilizes the *Proof-of-Data-IO* protocol to provide cryptographic assurances of any off-chain API calls (e.g., to LLM services), and the *Proof-of-Semantic* protocol to ensure the responses and actions are semantically aligned with expected behaviors. This dual-proof mechanism guarantees that each step of the workflow is both executed and interpreted correctly.

Workflows in VeriFlow are designed via an intuitive graphical user interface (GUI), allowing users to visually assemble nodes into directed acyclic graphs (DAGs), similar to existing AI orchestration systems like Dify. Nodes represent diverse functionalities, including data retrieval, model inference, on-chain state updates, off-chain computation, and inter-agent communication. The GUI generates a JSON-based specification of the workflow, which is then used for on-chain deployment.

Deployment of a Nucleus onto the Verisense Network is streamlined, typically executable via a single user command or action. Once deployed, the Nucleus operates as a persistent, verifiable agent on the network, capable of interacting with both on-chain and off-chain components as defined by its workflow logic. To coordinate with external agents or services, VeriFlow employs the **Model Context Protocol (MCP)** [2024] for structured contextual data exchange and Google's **Agent-to-Agent (A2A)** [2025]communication protocol for orchestrating autonomous agent interactions . MCP standardizes context sharing across model or agent boundaries, while A2A provides a framework for direct agent messaging and coordination in a decentralized manner.

Additionally, VeriFlow offers optional **EigenLayer restaking** to enhance security for sensitive workflow operations. By leveraging Ethereum validators who stake assets via EigenLayer, VeriFlow provides an extra economic security layer for critical workflow components, such as financial transactions or governance decisions within a workflow. Restaking ensures that even if the native Verisense consensus were to be subverted, the staked economic value on Ethereum would act as collateral to deter and penalize malicious activities.

Algorithm 4 outlines the VeriFlow compilation process:

VeriFlow thus represents a comprehensive orchestration platform that combines intuitive graphical workflow design with the robust security and verifiability afforded by blockchain execution. Through the integration of Proof-of-Data-IO, Proof-of-Semantic, MCP, and A2A protocols, as well as optional EigenLayer restaking, VeriFlow facilitates decentralized AI-driven workflows characterized by high trustworthiness, verifiability, and interoperability.

VerifFlow node types can be summarized as a table:

Algorithm 4 Compilation of Workflow to Nucleus

Require: Workflow specification W with blocks $B_1, B_2, \ldots, B_n$ and directed connections forming a DAG.
Ensure: Deployed Nucleus instance $\mathcal{N}$ on Verisense network

for each block B_i in W (topological order) **do**
    Validate functionality and requirements of B_i
    Convert B_i into its on-chain representation (e.g., smart contract or reference)
    if B_i requires off-chain execution **then**
        Prepare Proof-of-Data-IO hooks for B_i
    end if
end for
Package compiled on-chain components and metadata into Nucleus $\mathcal{N}$
Deploy $\mathcal{N}$ onto Verisense blockchain
return $\mathcal{N}$

5.1 VeriFlow Node Types

Currently, there are several centralized workflow architectures. In our approach, we have retained these nodes while implementing comprehensive security measures to ensure their integrity within the system. Table 3 presents a comparison between existing centralized workflow capabilities and VeriFlow capabilities, along with security assurances. Table 4 outlines the new node types introduced in the decentralized agent workflow.

Table 3: Comparison between Exists Centralized Workflow and VeriFlow Capabilities with Security Assurances

Node Type	**Description (Centralized Workflow)**	**VeriFlow**	**Onchain Security Assurance**
Start	Initiates workflows with user inputs	Supported	Input validation, authentication, verified by Nucleus
End	Marks workflow termination	Supported	Output verification by Nucleus
Answer	Delivers intermediate conversational responses	Supported	Semantic validation by Nucleus
LLM	Invokes LLM for content generation	Supported	Proof-of-Data-IO & Proof-of-Semantic validation
Conditional Branch	Implements IF/ELSE logic	Supported	Integrity checking
Code Execution	Executes Solidity or WASM or python3 scripts securely	Supported	Sandbox isolation
List Operator	Manipulates and sorts lists/arrays	Supported	Data integrity verification
Iteration	Performs operations iteratively on collections	Supported	Bound checking
Parameter Extraction	Converts text into structured data via LLM	Supported	Proof-of-Semantic validation
HTTP Request	Sends external HTTP API requests	Supported	Proof-of-Data-IO validation
Tools	Integrates built-in or custom external tools	TODO	Nucleus intercalls

Table 4: The new node types introduced in the decentralized agent workflow.

Node Type	**Description (Centralized Workflow)**	**VeriFlow**	**Onchain Security Assurance**
On-chain Event Trigger	Initiates workflows based on blockchain events	Newly Added	On-chain event validation
Smart Contract Call	Invokes smart contract functions directly	Newly Added	Cryptographic verification, ABI validation
Wallet Interaction	Manages wallet operations (transaction signing, verification)	Newly Added	Cryptographic signature validation
Decentralized Oracle Logic	Retrieves decentralized oracle data securely	Newly Added	Oracle data validation and verification
Cross-chain Messaging	Supports synchronization across multiple blockchains	Newly Added	Cross-chain consensus mechanisms (EigenLayer)
Proof-of-Data-IO	Cryptographic verification of data authenticity	Newly Added	TLSNotary-based cryptographic proofs
Proof-of-Semantic	Ensures semantic alignment of AI outputs	Newly Added	AI-driven semantic validation protocols

5.2 VeriFlow Plugin

VeriFlow is a modular intelligent workflow system supporting various Web3 ecosystem components through plugins. Users can integrate different blockchain services as node plugins into workflows, enabling complex on-chain and off-chain coordinated processes. For example, a workflow might use an oracle plugin to obtain market prices, execute token swaps via a DEX plugin, and subsequently store transaction results using a storage plugin. Below we detail various categories of Web3 components integrable as VeriFlow plugins.

5.2.1 Decentralized Exchanges (DEX)

A DEX is a decentralized exchange that enables direct peer-to-peer digital asset transactions on-chain without centralized intermediaries. Through DEX plugins, VeriFlow workflows can automatically execute token swaps, provide liquidity, and perform other related functions, facilitating automated trading processes within decentralized finance (DeFi) scenarios. For instance, workflow nodes can invoke DEX plugins to buy or sell specific assets when certain conditions are met. Integrating DEX plugins typically involves interacting with the decentralized exchange's smart contracts through Web3 interfaces to submit transactions. The supported DEX plugins are listed in Table 5.

Table 5: DEX related Plugin

Project	**Core Functionality**	**Reason for Node Integration**	**Integration Method**	**VeriFlow Use Case**
Uniswap	AMM-based decentralized exchange enabling permissionless exchanges between any ERC-20 tokens.	Provides readily available on-chain liquidity, enabling workflows to automate token exchanges or pricing.	Interact via smart contracts (e.g., Router contract) or use official SDKs for trading with Uniswap liquidity pools.	Automated arbitrage bots, decentralized financial transaction flows (e.g., asset swapping upon condition triggers).
Curve	Decentralized exchange specializing in low-slippage trades of stablecoins and similar assets, optimizing transactions through specialized AMM curves.	Ideal for stablecoin exchanges within workflows, providing low slippage and deep liquidity.	Conduct transactions via smart contract interfaces or integrate using Curve's developer libraries for executing swaps.	Stablecoin exchanges in cross-border payment workflows, asset rebalancing in yield aggregation strategies.

5.2.2 Native Chain Services

Native chain services refer to the fundamental functionalities or infrastructures provided by public blockchains themselves, such as smart contract platforms, local cryptocurrency transactions, and consensus mechanisms. By integrating public chains as VeriFlow plugin nodes, on-chain operations or state queries can be executed directly within workflows. Integrating mainstream blockchain nodes allows VeriFlow to support various on-chain tasks such as contract deployment, transaction sending, or account balance checking, seamlessly connecting workflows with blockchain applications. Table 6 lists the supported native chain plugins.

Table 6: Native Chain related Plugin

Project	**Core Functionality**	**Reason for Node Integration**	**Integration Method**	**VeriFlow Use Case**
Ethereum	General-purpose smart contract platform supporting complex decentralized application development and operation, with a rich ecosystem.	Workflows need interaction with the most widely used blockchain applications, such as invoking DeFi protocols or issuing NFTs, requiring Ethereum node support.	Interaction via Ethereum full nodes or Web3 RPC interfaces (e.g., Infura), utilizing libraries like Web3.js or ethers.js to send transactions and query states.	Automatic execution of Ethereum transactions in workflows (e.g., payments, swaps), and listening to on-chain events to trigger subsequent processes.
BSC (Binance Smart Chain)	Ethereum-compatible smart contract blockchain with fast transactions and low fees, suitable for large-scale applications.	As a high-performance blockchain, BSC accelerates frequent on-chain operations in workflows, reducing execution costs.	Connect via BSC RPC nodes and interact with its smart contracts. Due to EVM compatibility, Ethereum integration methods can be reused.	Nodes handling high-volume user transactions such as in-game item trades and processing numerous micro-transactions.
Solana	High-performance blockchain employing unique Proof of History consensus, supporting extremely high TPS and low latency.	Suitable for tasks requiring real-time responsiveness or high transaction volumes; integrating Solana enhances workflow performance.	Communication via Solana RPC interface or Web3 SDK (e.g., solana-web3.js), submitting transactions or invoking on-chain programs.	Workflow nodes requiring high-speed processing, such as real-time data recording or executing on-chain logic for competitive games.

5.2.3 Cross-chain Communication

Cross-chain communication components enable message or asset transfer between different blockchain networks, breaking barriers between chains. VeriFlow's cross-chain plugins facilitate actions triggered on one blockchain to execute results on another. For example, workflow nodes can relay an event notification from chain A to chain B or transfer tokens between different chains. Cross-chain communication plugins ensure coordinated and consistent execution across multiple blockchain interactions within workflows. The supported cross-chain communication plugins are detailed in Table 7.

5.2.4 Oracles

Oracle components provide blockchains with off-chain or cross-chain data input. As smart contracts inherently cannot access real-world off-chain data, oracle plugins allow VeriFlow to obtain external information such as prices, weather conditions, and event outcomes, which can be utilized in workflows for decision-making purposes. With oracles, workflows can execute based on real-time real-world data, thus expanding the scope of blockchain applications. Additionally, oracles facilitate data transfer between different blockchains, acting as information bridges in cross-chain scenarios. Supported oracle plugins are summarized in Table 8.

Table 7: Cross-chain Communication related Plugin

Project	**Core Functionality**	**Reason for Node Integration**	**Integration Method**	**VeriFlow Use Case**
Polkadot	Multi-chain architecture based on relay chains enabling cross-chain message passing (XCM) between parachains and shared security.	Necessary for workflows involving multiple chains within the Polkadot ecosystem (e.g., executing tasks on different parachains), leveraging Polkadot's cross-chain coordination.	Operate Polkadot relay chain nodes and use Polkadot APIs (e.g., Polkadot.js) to send XCM cross-chain messages, scheduling actions on parachains within workflows.	Supply chain finance scenarios where businesses deploy operations on different parachains, VeriFlow plugins facilitate instruction transmission and result aggregation between chains.
Cosmos IBC	Cosmos ecosystem's Inter-Blockchain Communication protocol, enabling trusted message and token exchange among independent blockchains (Zones).	Provides reliable communication channels within Cosmos's multi-chain ecosystem for workflows requiring services across different Zones.	Connect Cosmos SDK chain nodes and construct cross-chain transactions using IBC interfaces, enabling inter-zone interactions via relays like Cosmos Hub (Gaia).	Multi-chain DeFi operations, such as collateralizing assets on one chain and borrowing on another, with workflows coordinating these operations via IBC plugins.
LayerZero	Decentralized cross-chain communication infrastructure using oracles and relayers to transmit messages directly between heterogeneous chains.	Connects major blockchains, allowing workflow nodes to send instructions or transfer assets across different chains using LayerZero.	Deploy LayerZero endpoint contracts on source and destination chains, leveraging its APIs to transmit messages securely via a decentralized relay network.	Cross-chain arbitrage bots: detecting price differences on the source chain and executing trades on the destination chain via LayerZero plugins.

Table 8: Oracles related Plugin

Project	**Core Functionality**	**Reason for Node Integration**	**Integration Method**	**VeriFlow Use Case**
Chainlink	Decentralized oracle network providing data feeds such as price quotes, financial data, and randomness, widely used across various smart contracts.	Many DeFi protocols and contract logic require reliable external data like prices. A Chainlink plugin ensures authoritative and tamper-resistant data for workflows.	Fetch data published on-chain through Chainlink's data feed contracts (e.g., price feeds) or run Chainlink nodes subscribing to specific data sources to write data on-chain.	Checking collateral prices in automated liquidation processes, triggering on-chain reward or penalty mechanisms based on real-world events (e.g., sports results).

5.2.5 Restaking

Restaking refers to leveraging already-staked assets (such as staked tokens) in one network to provide additional security or services in another protocol. These components extend the base blockchain's security trust to new modules or blockchain services. Through restaking plugins, VeriFlow can utilize the security of the primary blockchain to safeguard customized applications. For example, workflows can borrow Ethereum validator stakes to operate an off-chain service node, thus reducing trust costs for such services. Supported restaking plugins are listed in Table 9.

Table 9: Restaking related Plugin

Project	**Core Functionality**	**Reason for Node Integration**	**Integration Method**	**VeriFlow Use Case**
EigenLayer	Ethereum ecosystem restaking protocol allowing Ethereum stakers to re-use staked ETH to provide security for third-party modules (e.g., oracles, data availability layers).	By integrating EigenLayer, VeriFlow nodes can benefit from Ethereum mainnet security. Custom service nodes connected to EigenLayer gain enhanced trustworthiness.	Interact with EigenLayer contracts to register nodes as EigenLayer services and accept restaked ETH as collateral, or monitor node statuses and stakes via EigenLayer's API.	Constructing highly secure cross-chain bridges or oracle nodes using restaked ETH to ensure the security and reliability of critical workflow steps such as cross-chain transaction confirmations.
SatLayer (Babylon)	Bitcoin restaking protocol that leverages staked BTC to secure other protocols (providing BVS security); programmable slashing enabled via smart contracts, extending the security reach of BTC	Enhances PoS chains and DApps with Bitcoin's economic security; enables nodes to participate in Bitcoin Validation Services (BVS) and benefit from Bitcoin-level finality and punitive measures	Integrated via Babylon sidechain: run Babylon nodes or smart contracts, monitor BTC staking status (via SPV/time-lock proofs), and trigger slashing transactions to penalize misbehavior	Using Bitcoin's immutability for cross-chain state validation, e.g., submitting PoS chain snapshot hashes to Bitcoin (timestamp proof); triggering BTC collateral slashing upon detection of validator misbehavior to secure cross-chain events
Solayer (Solana)	Native restaking network on Solana, converting staked SOL (or LSTs) into sSOL to support additional verification services (AVS); strengthens on-chain application security and provides extra rewards	Expands validator responsibilities by allowing nodes to participate in additional services (e.g., Rollup nodes, DePIN infrastructure) through restaking, thus enhancing service reliability and economic incentives	Natively integrated within the Solana network: run Solayer protocol nodes or delegate to Solayer operators without cross-chain operations, directly staking through Solana's delegation mechanisms	In VeriFlow, used to verify the validity of Solana subchains or special execution layers: e.g., verifying Rollup block production or monitoring infrastructure service integrity; ensuring critical workflow security under the Solana security umbrella

5.2.6 Storage

Storage components offer decentralized off-chain storage services for large amounts of data, files, or backup information. Since blockchains are unsuitable for directly storing large files, storage plugins enable VeriFlow workflows to read and write data such as user profiles, transaction records, and multimedia files to decentralized storage networks. This ensures data durability and tamper resistance while reducing the load on the main chain. Typical distributed storage solutions include content-addressed file systems and long-term blockchain storage. Supported storage plugins are outlined in Table 10.

5.2.7 Fingerprint

VeriFlow's fingerprint support leverages the OML [2024] framework to introduce cryptographic model fingerprinting capabilities in the network. This integration ensures that all AI models and their outputs can be verified for rightful ownership and provenance, thereby preventing unauthorized model use in decentralized applications. By verifying

Table 10: Storage related Plugin

Project	Core Functionality	Reason for Node Integration	Integration Method	VeriFlow Use Case
IPFS	Peer-to-peer distributed file storage and transfer protocol storing data using content addressing, eliminating central servers.	Provides highly reliable decentralized storage for medium-sized files or configurations within workflows, allowing precise retrieval through content hashes.	Run IPFS nodes to upload files via its API to obtain content hashes or retrieve files using hashes; alternatively, use public gateways for read/write operations.	Uploading data analysis results to IPFS, storing the hash on-chain in subsequent workflow steps; distributing shared configuration files across multiple collaborating nodes.
Arweave	Decentralized permanent storage network leveraging blockchain technology, offering perpetual data storage with a one-time fee.	Ideal for permanently storing crucial data such as important documents or notarized records, ensuring immutability and long-term availability.	Submit data files to the Arweave network via its SDK or HTTP interface, pay a one-time fee, and obtain a content address for future access.	Archiving notarized documents or certificates permanently on Arweave for future authenticity verification; storing historical transaction records for auditing purposes.

model outputs or the models themselves against registered fingerprints, VeriFlow nodes help maintain a secure AI model marketplace and block stolen models from being onboarded into decentralized applications.

Table 11: Fingerprint related Plugin

Project	Core Functionality	Reason for Node Integration	Integration Method	VeriFlow Use Case
OML (Open, Monetizable, and Loyal AI)	Model fingerprinting, ownership validation, provenance tracking.	Ensure traceable AI model ownership and prevent unauthorized model use.	Deploy OML-compatible fingerprint verifiers in VeriFlow nodes; verify model outputs or models against registered fingerprints.	AI model marketplace integration; preventing stolen models from being onboarded into decentralized applications.

5.2.8 Modular Blockchain

Modular blockchain separates functionalities (consensus, data availability, execution) across different layers or chains, enabling specialized blockchain services. VeriFlow can utilize these modular blockchain components, combining their strengths to enhance scalability and customization. Table 12 presents the supported modular blockchain plugins.

5.3 dMCP & dA2A Protocol

This section presents decentralized extensions to the existing Model Context Protocol (MCP) and Agent-to-Agent (A2A) protocol, termed dMCP and dA2A, respectively. These enhancements facilitate agent orchestration in decentralized networks, adding critical new functionality while collaborating with Google's team to extend the original A2A protocol.

5.3.1 Core Concepts

- **dMCP: Decentralized Model Context Protocol:** dMCP transforms centralized orchestration into decentralized coordination among multiple autonomous agents. Unlike traditional MCP, dMCP leverages peer-to-peer

Table 12: Modular Blockchain related Plugin

Project	Core Functionality	Reason for Node Integration	Integration Method	VeriFlow Use Case
Celestia	Provides blockchain data availability and consensus layers without executing smart contracts, securely publishing transaction data.	Efficient data availability in workflows involving extensive data or transaction proof aggregation.	Submit data blocks via Celestia API or validate published data proofs in collaboration with execution layers.	Sharding or rollup workflows: batch transaction data submitted to Celestia for availability before execution on another chain.
Fuel	Modular execution layer enhancing throughput via UTXO model and parallel transactions, functioning as an independent execution layer for Ethereum.	High-performance smart contract execution environments with secure finalization on the main chain.	Deploy and call contracts on Fuel network via SDK (e.g., Fuel Rust SDK), synchronize state or finalize results with Ethereum mainnet.	Intensive computation tasks executed on Fuel, with results settled on the main chain, boosting workflow efficiency.

(P2P) interactions, enabling distributed task decomposition, allocation, and synchronization without a central orchestrator.

- **dA2A: Decentralized Agent-to-Agent Protocol:** dA2A enhances traditional A2A protocols by eliminating reliance on centralized infrastructures for agent discovery and communication, adopting a fully decentralized P2P model. It includes distributed service discovery, dynamic peer connection, and decentralized authentication mechanisms.

5.3.2 Comparison with Traditional Protocols

- **Architecture**: dMCP and dA2A adopt decentralized peer-to-peer networks, contrasting with the centralized orchestrator and client-server models.
- **Information Structure**: Enhanced message formats including task dependency graphs, distributed state synchronization fields, and shared contextual data.
- **Security Mechanisms**: Implements distributed trust models using decentralized identifiers (DIDs), end-to-end encryption, and digital signatures, moving beyond traditional centralized PKI-based trust models.

5.3.3 Adaptation to Decentralized Networks

Protocols utilize P2P network layers and distributed ledgers for agent discovery, task assignment, and result synchronization, ensuring robustness and scalability. Distributed ledgers (e.g., blockchain) optionally provide persistent global state management.

5.4 Collaboration with Google

Our team collaborates closely with Google to enhance the original A2A protocol. Key modifications include compatibility with decentralized network topologies, additional metadata fields for P2P endpoints, and decentralized authentication methods. Current progress includes a prototype demonstrating these capabilities within a libp2p-based implementation.

5.5 Protocol Extensions

5.5.1 Additional Protocol Fields

- **NodeID**: Unique identifiers for agent nodes.
- **Digital Signatures**: Ensuring authenticity and integrity.
- **Task Topology**: Expresses complex workflows and dependencies.
- **Confirmation and Feedback Fields**: Explicit acknowledgments and confirmations.

- **Timestamps and Sequence Numbers**: Preventing replay attacks and maintaining message order.

5.5.2 Enhanced Message Flows

- **Agent Discovery**: Decentralized dynamic discovery.
- **Task Negotiation**: Multi-stage negotiation for task assignment.
- **Distributed Execution and Synchronization**: Real-time updates and collaboration.
- **Result Consolidation and Verification**: Decentralized verification of task completion.

The dMCP and dA2A protocol extensions successfully decentralize agent orchestration, ensuring scalability, robustness, and enhanced security. Collaboration with Google and community engagement will further validate and refine these protocols.

6 Example: BTC Quantitative Trading Bot using VeriFlow

In this section, we demonstrate how the **VeriFlow** framework can be leveraged to build a fully functional Bitcoin (BTC) quantitative trading robot. The robot consists of three interconnected modules: Fundamental Analysis, Technical Analysis, and the Final Decision Agent.

6.1 Fundamental Analysis Module

The fundamental analysis module (Fig. 4 and Alg. 5) is responsible for collecting and structuring news-based sentiment data.

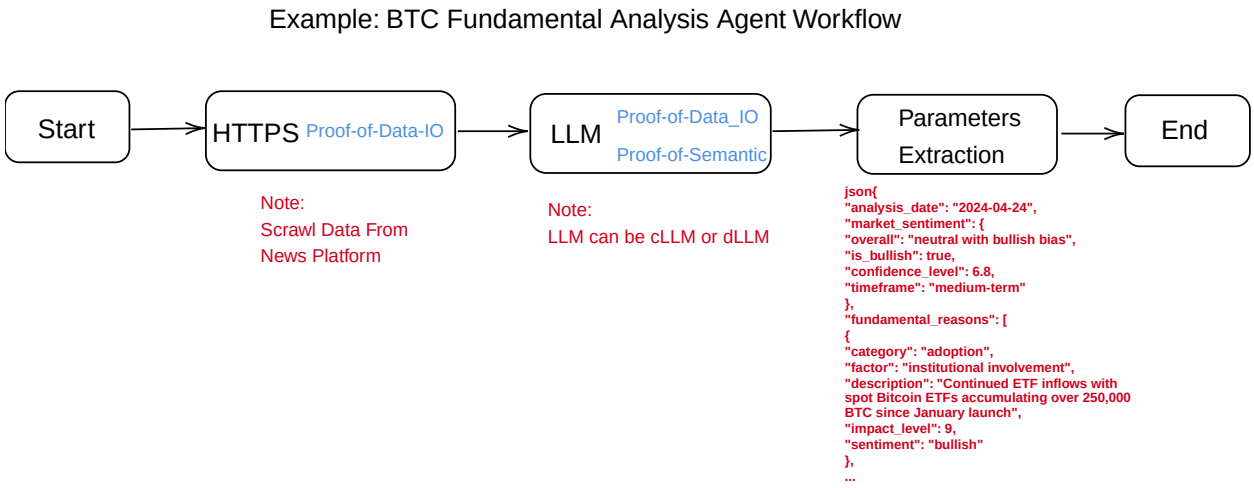

Figure 4: BTC Fundamental Analysis Module

This module begins by securely fetching news data via HTTPS, utilizing VeriFlow's decentralized Model Context Protocol (dMCP) and Proof-of-Data-IO to ensure data authenticity. An LLM (either centralized cLLM or decentralized dLLM) is employed to parse this data, producing structured sentiment indicators and insights verified through Proof-of-Semantic. The outcome is a structured JSON format detailing sentiment analysis and fundamental market drivers.

The following pseudocode describes the fundamental analysis workflow:

6.2 Technical Analysis Module

The technical analysis module (Fig. 5 and Alg. 6) utilizes market data, user preferences, and quantitative systems to determine potential trading positions.

Algorithm 5 Fundamental Analysis Module

1: Fetch latest BTC news via HTTPS (secured with Proof-of-Data-IO)
2: Analyze content using LLM (Proof-of-Semantic)
3: Extract sentiment parameters (confidence level, market sentiment)
4: Format and output structured JSON data

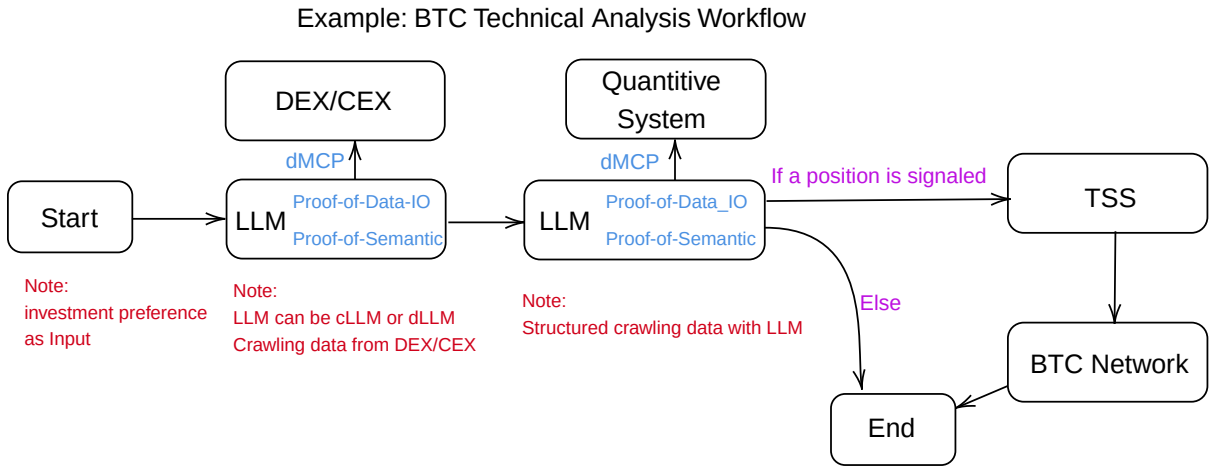

Figure 5: BTC Technical Analysis Module

This module initiates by capturing user-defined investment preferences. The LLM then securely gathers real-time market data from decentralized and centralized exchanges via dMCP, with each input verified by Proof-of-Data-IO. Additionally, quantitative analysis signals are integrated to enrich the decision context. After evaluating these inputs (secured by Proof-of-Semantic), the module decides whether to initiate a BTC transaction. If conditions are met, transactions are securely executed through a Threshold Signature Scheme (TSS).

Technical analysis workflow pseudocode is as follows:

Algorithm 6 Technical Analysis Module

1: Retrieve user investment preferences
2: Fetch market data via LLM from DEX/CEX (Proof-of-Data-IO)
3: Integrate quantitative analysis signals
4: Evaluate trading conditions (Proof-of-Semantic)
5: **if** trading conditions met **then**
6: Execute trade via Threshold Signature Scheme
7: Broadcast transaction to BTC network
8: **else**
9: Take no action
10: **end if**

6.3 Final Decision Module (Orchestration)

The final orchestration module (Fig. 6 and Alg. 7) combines fundamental and technical analyses to render a comprehensive trade decision.

Using decentralized agent-to-agent communication (dA2A), this module obtains the structured results from both the fundamental and technical agents. The LLM within this module evaluates the consistency and alignment of both inputs, making a final decision that is secured by both Proof-of-Data-IO and Proof-of-Semantic. Decisions can result in trade execution or veto, depending on predefined strategic rules and data coherence.

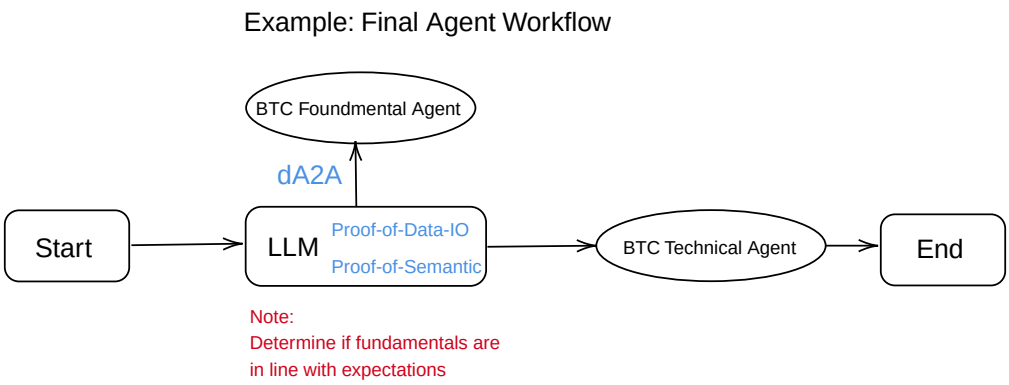

Figure 6: BTC Final Decision Agent

The orchestration pseudocode is illustrated below:

Algorithm 7 Final Decision Module (Orchestration)

```
Retrieve results from Fundamental and Technical agents (via dA2A)
Verify integrity of inputs (Proof-of-Data-IO)
Evaluate alignment with LLM (Proof-of-Semantic)
if analyses aligned and strategic rules satisfied then
    Confirm trade execution
else
    Reject trade
end if
Output final decision with rationale
```

Throughout all modules, VeriFlow's Proof-of-Data-IO and Proof-of-Semantic mechanisms significantly enhance trustworthiness, ensuring the integrity and logical coherence of the robot's decision-making processes.

7 Future Work

Future research directions will focus on enhancing the decentralized agent-to-agent communication protocols. Specifically, we will collaborate with Google to design and refine a fully decentralized Agent2Agent (A2A) protocol, aiming for increased interoperability, scalability, and robustness in dynamic decentralized environments. Additional exploration will involve the integration of advanced cryptographic techniques, including further optimizations of Fully Homomorphic Encryption (FHE) and zero-knowledge proofs, to improve data privacy and security. Extensive performance evaluations and real-world deployment scenarios will also be conducted to validate and optimize system performance under various operational conditions.

8 Conclusion

This paper has proposed the Verisense Network as a robust solution for decentralized agent execution and multi-agent orchestration using an active blockchain paradigm. By moving beyond traditional static blockchain functionalities, Verisense Network offers autonomous agents the ability to engage dynamically and securely. The proposed decentralized MCP, semantic-based consensus mechanisms, and active orchestration architecture collectively address critical challenges inherent to decentralized systems, including coordination complexity, security, and efficient resource utilization. Through practical implementations, Verisense Network demonstrates significant potential for broader applications in decentralized AI ecosystems, IoT networks, and autonomous decision-making platforms.

References

0G Labs (2024). 0G: The first decentralized AI operating system. Project documentation. `https://github.com/0glabs/awesome-0g`.

Ambient Finance (2024). Ambient litepaper. `https://ambient.xyz/litepaper`. Accessed: 2025-04-23.

Anthropic (2024). Introducing the model context protocol. `https://www.anthropic.com/news/model-context-protocol`. Accessed: 2025-04-23.

Castro, M. and Liskov, B. (1999). Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186. USENIX Association.

Cheng, Z., Contente, E., Finch, B., Golev, O., Hayase, J., Miller, A., Moshrefi, N., Nasery, A., Nailwal, S., Oh, S., Tyagi, H., and Viswanath, P. (2024). OML: Open, monetizable, and loyal AI. Cryptology ePrint Archive, Paper 2024/1573.

Cheon, J. H., Kim, A., Kim, M., and Song, Y. (2017). Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology – ASIACRYPT 2017*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer.

Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. (2020). Tfhe: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(3):709–763.

DAGWorks Inc. (2023). DAGWorks: A platform for reliable AI pipelines and workflows. Online documentation. `https://dagworks.io`.

Desmedt, Y. and Frankel, Y. (1989). Threshold cryptosystems. In *Advances in Cryptology–CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 307–315. Springer.

Dify Team (2025). Dify: The innovation engine for generative ai applications. `https://dify.ai`. Accessed: 2025-04-23.

Douceur, J. R. (2002). The sybil attack. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 251–260. Springer.

Gentry, C. (2009). Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, pages 169–178. ACM.

Goldwasser, S., Micali, S., and Rackoff, C. (1985). The knowledge complexity of interactive proof systems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC)*, pages 291–304. ACM.

Google Cloud (2025). Announcing the agent2agent protocol (a2a). `https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/`. Accessed: 2025-04-23.

Kalka, M. and Kirejczyk, M. (2024). A comprehensive review of tlsnotary protocol. `https://arxiv.org/abs/2409.17670`. Accessed: 2025-04-23.

Merkle, R. C. (1988). A digital signature based on a conventional encryption function. In *Advances in Cryptology–CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. White paper. `https://bitcoin.org/bitcoin.pdf`.

Primus Labs (2025). Primus labs: Verifiable data infrastructure with zktls and zkfhe. `https://primuslabs.xyz`. Accessed: 2025-04-23.

Rao, Y. (2020). Bittensor: A peer-to-peer intelligence market. `https://bittensor.com/whitepaper`. Accessed: 2025-04-23.

Reimers, N. and Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks.

Ritzdorf, H., Wüst, K., Gervais, A., Felley, G., and Capkun, S. (2018). Tls-n: Non-repudiation over tls enabling ubiquitous content signing. In *25th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society.

SECBIT Labs (2019). zkpod: A practical decentralized system for data exchange. `https://secbit.io/zkPoD-node/paper.pdf`. Accessed: 2025-04-23.

Sunscreen Labs (2025). Sunscreen: Privacy engine of the new web. `https://sunscreen.tech`. Accessed: 2025-04-23.

Team, E. (2023). EigenLayer: Decentralized trust through ethereum restaking. White paper (online). `https://eigenlayer.xyz`.

Verisense Network (2024). Verisense network: Active blockchain with fhe-enabled validation-as-a-service. https://github.com/verisense-network/papers/blob/main/whitepaper/verisense_whitepaper_20240818.pdf. Accessed: 2025-04-23.

Walters, S., Gao, S., Nerd, S., Da, F., Williams, W., Meng, T.-C., Han, H., He, F., Zhang, A., Wu, M., Shen, T., Hu, M., and Yan, J. (2025). Eliza: A Web3-friendly AI agent operating system. *arXiv preprint arXiv:2501.06781*.

Wetzel, B. (2023). Tls oracles (zktls): Liberating private web data with cryptography. https://bwetzel.medium.com/tls-oracles-liberating-private-web-data-with-cryptography-e66e5fad7c34. Accessed: 2025-04-23.

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, (151):1–32. https://ethereum.github.io/yellowpaper/paper.pdf.

Wood, G. (2016). Polkadot: Vision for a heterogeneous multi-chain framework. White Paper. https://polkadot.network/PolkaDotPaper.pdf.

Zhang, F., Maram, S. K. D., Malvai, H., Goldfeder, S., and Juels, A. (2020). Deco: Liberating web data using decentralized oracles for tls. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1919–1938. ACM.

Zhang, Y., Yan, X., Tang, G., and Wang, H. (2024). Monadring: A lightweight consensus protocol to offer validation-as-a-service to avs nodes. https://arxiv.org/abs/2408.16094. Accessed: 2025-04-23.