Know Your Agent (KYA): A Decentralized Identification Layer for Agentic AI

Micky Multani
https://github.com/mickymultani
micky.multani@gmail.com

July 23, 2025

Abstract

The rapid proliferation of autonomous AI agents necessitates a standardized, secure, and verifiable method for identification and authorization. Current identity and access management (IAM) solutions, designed primarily for human users, are ill-equipped to handle the unique requirements of agentic AI. This whitepaper introduces the "Know Your Agent" (KYA) system, a novel decentralized identity framework that leverages the immutability of the Solana blockchain and the version control capabilities of GitHub. The KYA system provides each AI agent with a unique, cryptographically verifiable identity, enabling secure commerce, granular permissioning, and transparent interactions across diverse platforms. We detail the system's architecture, including its GitHub-based profile management, Solana NFT integration, robust security protocols, and strategies for schema evolution and error handling. By establishing a native, auditable identity layer for AI agents, KYA paves the way for a trustworthy and highly efficient agentic future in the digital economy.

Contents

1 Introduction

The future of Artificial Intelligence is undeniably agentic. As AI models evolve from passive tools into autonomous entities capable of initiating actions, engaging in commerce, and interacting with complex digital ecosystems, the fundamental requirement for a robust and verifiable identity emerges as paramount. Just as "Know Your Customer" (KYC) regulations are vital for financial systems dealing with human entities, a "Know Your Agent" (KYA) framework is indispensable for ensuring accountability, security, and trust in the burgeoning world of AI agents.

Current identity and access management (IAM) solutions, largely conceived for human users and traditional software applications, are inherently insufficient for the dynamic, often ephemeral, and highly autonomous nature of AI agents. Challenges include:

- **Lack of Standardized Identification:** No unified method exists for AI agents to assert and prove their identity across diverse platforms.
- **Verification Gaps:** Difficulties in verifying an agent's true origin, its authorized scope, and the legitimacy of its intentions.
- **Security Vulnerabilities:** Susceptibility to impersonation, unauthorized access, and fraud due to weak or absent identity layers.
- **Limited Permissioning:** Inability to assign granular, context-aware, and auditable permissions specifically tailored to an agent's capabilities and purpose.
- **Auditability Challenges:** Difficulty in tracing and attributing autonomous actions performed by AI agents back to their owners or authorizing entities.

These limitations directly impede the seamless integration of AI agents into critical domains such as e-commerce, financial services, data management, and automated task execution. The absence of a trust layer inhibits their participation in commerce, restricts data access, and complicates compliance.

This whitepaper introduces the Know Your Agent (KYA) system, a novel architectural solution designed to provide a secure, decentralized, and verifiable identity layer for AI agents. By combining the strengths of the Solana blockchain for immutable identity anchoring and GitHub for transparent, version-controlled profile management, KYA addresses the shortcomings of existing paradigms. The proposed system aims to foster a new era of trust and interoperability, enabling AI agents to contribute effectively and securely to the global digital economy.

2 The KYA System: A Decentralized Identity Framework for AI Agents

Our KYA system establishes a robust and verifiable identity for every AI agent, leveraging the transparency and immutability of blockchain technology alongside familiar version control principles. At its core, the system utilizes a unique identity for each AI agent, represented as a **fork of a base repository on GitHub**. This GitHub repository serves as the agent's digital passport, containing all its essential KYA data. This fork is then referenced on the

Solana blockchain as a **Non-Fungible Token (NFT)**. The NFT's address itself, or a specific field within its metadata, serves as the agent's unique **KYA hash**. This KYA hash acts as a publicly verifiable and immutable pointer to the agent's detailed identity on GitHub, allowing any service or platform to easily verify the agent's identity and permissions, ensuring secure and trustworthy interactions.

2.1 Architectural Deep Dive: GitHub Repository Structure

The GitHub repository for each AI agent is meticulously structured to accommodate comprehensive KYA data, ensuring both flexibility and security. The design facilitates easy version control, transparent auditing, and secure handling of sensitive information. Each AI agent's KYA profile originates from a fork of a standardized base schema repository, ensuring consistency while allowing for individual customization and independent evolution.

The repository will contain the following key files and directories:

- `agent.json`: This is the core configuration file for the AI agent, providing fundamental identity metadata.
  - `name`: The human-readable name of the AI agent (e.g., "FoodieBot", "SalesAssistBot").
  - `type`: Categorization of the agent's entity type (e.g., "individual", "enterprise", "sub-agent"). This field is crucial for applying different schema rules and access policies, as individual agents might have personal payment methods, while enterprise agents might be tied to corporate AD groups.
  - `solana_nft_address`: The unique Solana NFT address associated with this agent's KYA profile. This acts as the direct on-chain link to its verifiable identity.
  - `owner_id`: A reference to the creator or owning entity. For individual agents, this could be a user ID; for enterprise agents, an organization ID, department ID, or a reference to an Active Directory (AD) group.
  - `creation_date`: A timestamp indicating when the agent's KYA profile was initially created.
  - `purpose`: A brief, high-level description of the agent's intended function or primary capabilities (e.g., "Automates food ordering," "Assists sales representatives with CRM data queries").
- `data/`: This directory houses agent-specific operational information, structured for clarity and granular access control.
  - **Subdirectories for Categorization**: To ensure organized storage and facilitate modular updates, the `data/` directory will contain subdirectories. Common categories might include:
    - `capabilities/`: Defines the agent's inherent functionalities (e.g., `api_access.json`, `language_processing.json`, `transaction_capabilities.json`). For instance, `api_access.json` might list external APIs the agent is designed to interact with.
    - `limitations/`: Specifies constraints on the agent's behavior or scope (e.g., `geographical_restriction` `time_restrictions.json`).

    * `authorized_actions/`: Lists the precise actions the agent is permitted to perform, often mapped to specific service endpoints (e.g., `ordermgmt.json` for food ordering, `crm_query.json` for database lookups). This can be a detailed list of authorized method calls or operations.
    * `payment_profiles/`: Contains references to authorized payment methods. Crucially, this file will **not store raw payment details**. Instead, it will hold encrypted references (e.g., a tokenized card number from a payment gateway, a wallet address) or unique identifiers linked to secure, external payment vaults. This is designed to separate sensitive data from the public KYA profile.
    * `spend_limits/`: Defines financial constraints that apply to the agent (e.g., `daily_limit.json`: $500, `transaction_limit.json`: $100, `authorized_categories.json`: ["food", "travel", "software_subscriptions"]). These limits provide a crucial guardrail for financial autonomy.
    * `authorized_platforms/`: Lists specific platforms or services the agent is explicitly authorized to interact with, possibly including their base URLs or API endpoints. This prevents an agent from attempting to act on unauthorized services.
    * `consent_management/`: Outlines data usage policies and user consent preferences relevant to the agent's operation. This is crucial for GDPR, CCPA, and other privacy regulations.
  - **JSON Files for Data Storage**: All data within these subdirectories will be stored in JSON format for easy parsing, validation, and programmatic access.

- `schema/`: This directory is fundamental for ensuring consistency and integrity. It stores JSON Schema definitions that rigorously enforce the structure, data types, and validation rules for the `data/` directory content and the `agent.json` file.
  - **Enforcement**: Any updates to an agent's GitHub KYA profile will be validated against these schemas. Invalid updates will be rejected by the KYA platform's processing pipeline.
  - **Type Specific Schemas**: Different schemas can exist within this directory for different agent `type`s (e.g., `individual_agent_schema.json`, `enterprise_agent_schema.json`), allowing for tailored fields and validation rules.

- `credentials/`: This directory is designed for referencing and securely accessing highly sensitive, encrypted credentials and secrets. It is paramount that these files are **never stored in plain text** directly within GitHub and are always encrypted using robust cryptographic methods.
  - **Encrypted References/Pointers**: Files in this directory will contain encrypted references to credentials (e.g., encrypted API tokens, encrypted login details for specific websites, encrypted pointers to secure key vaults or Hardware Security Modules - HSMs). The raw, unencrypted secrets are never committed to GitHub.
  - **Secure Access Mechanism**: Access to these encrypted credentials will be strictly controlled. Agents will trigger a secure access protocol (detailed in Section 6.4) to decrypt and utilize these credentials only when necessary for a specific, authorized action. This typically involves a secure element (e.g., user's KYA app, enterprise KMS/HSM) holding

the decryption key.
- **Example**: A file like `credentials/api_keys.enc` might contain a securely encrypted blob of API keys for services the agent uses. An agent would never see the raw key; its managing application would decrypt it on demand for a specific, authorized API call.

Versioning and Updates: The intrinsic versioning capabilities of Git are leveraged for managing updates to an agent's KYA profile. Every modification to the GitHub repository constitutes a new commit, creating an immutable, auditable history of the agent's identity and permissions. Importantly, each significant update to the KYA data will trigger an update to the associated Solana NFT's metadata (specifically, the `GitHub Commit Hash` attribute), ensuring the publicly verifiable on-chain record always reflects the latest, verified state of the agent's profile. This tight coupling guarantees that any external platform checking the KYA NFT can always fetch the precise, authorized version of the agent's capabilities.

2.2 Technical Implementation on Solana and Verification Flow

The KYA system's on-chain functionality is powered by a Solana program (smart contract) written in Rust, leveraging the Solana Program Library (SPL) standards, particularly for Non-Fungible Tokens (NFTs). This program will manage the minting, ownership, and metadata updates of the KYA NFTs, serving as the immutable root of trust for AI agent identities.

2.2.1 Core Program Responsibilities

The KYA Solana Program (`kya_program`) will have the following primary responsibilities:

- **NFT Minting and Management:** Responsible for creating unique KYA NFTs, each representing a distinct AI agent identity. This involves initializing mint accounts and token accounts according to SPL standards.
- **Ownership Tracking:** Maintaining verifiable ownership of KYA NFTs by linking them to specific user or organization Solana public keys. Transfers of NFT ownership on-chain directly reflect a change in the agent's control.
- **Metadata Referencing:** Storing a secure, immutable reference within the NFT's metadata that points to the agent's detailed KYA profile on GitHub. This reference will typically include the GitHub repository URL and a specific commit hash for version integrity.
- **Verification Interface:** Providing public functions and data structures that allow other Solana programs (on-chain) or off-chain services (via RPC calls) to query an NFT's metadata, verify its authenticity, and retrieve the pointers to its off-chain profile.
- **Permission Updates (Indirect):** Facilitating the update of the NFT's metadata when the off-chain KYA GitHub profile changes. This ensures the on-chain immutable record always points to the latest authorized version of the agent's KYA data.

2.2.2 Key Accounts and Data Structures

The `kya_program` will interact with and manage several types of Solana accounts, which are foundational to its operation:

- `KYA_NFT_Mint_Account` **(SPL Token Mint Account):** This is the core account defining the unique NFT.
  - **Data Stored**: 'supply' (always 1 for an NFT), 'decimals' (always 0 for an NFT), 'mint$_{a}uthority$'($thepublic$
    $OwnedbytheSPLTokenProgram.$
  - **Role**: Represents the unique KYA identity of a single AI agent. Each new agent requires a new 'Mint$_{A}ccount$'.
- `KYA_NFT_Token_Account` **(SPL Token Account):** An SPL Token Account that holds the KYA NFT for a specific user or organization.
  - **Data Stored**: 'mint' (a reference to the 'KYA$_{N}FT_{M}int_{A}ccount$'$itholdstokensfrom$), '$owner$'($thepublicke$
    $OwnedbytheSPLTokenProgram, butits$'$owner$'$fielddesignatestheactualcontrolleroftheNFT.$
- **Role**: Proves that a specific user/organization's 'Pubkey' possesses the KYA NFT for a given agent, thereby asserting control over that agent's KYA identity.
- **Program Derived Addresses (PDAs):** The `kya_program` will extensively use PDAs for various internal state management and to enable the program itself to "sign" for certain operations without requiring a private key. PDAs are deterministic and cannot be directly controlled by external users, making them ideal for program-controlled assets and configurations.
  - `KYA_Mint_Authority_PDA`: A PDA specifically designed to act as the 'mint$_{a}uthority$'$fornewKYANFTs.$
  - **Seeds**: '["kya$_{m}int_{a}uthority$", $program_{i}d$]'($derivedfromafixedstringandtheprogram'sownID$).**Purpose** :
    $Thisallowsthe$`kya_program`$toprogrammaticallymintheinitialNFTintotheagentowner'saccount.Afterthe$
    $mintable, ensuringitsuniqueness.$
- `KYA_Config_PDA`: A single PDA to store global configuration data for the entire KYA system.
  - **Seeds**: '["kya$_{c}onfig$", $program_{i}d$]'**Purpose** : $Canstoreimmutableormutablesystem-wideparameterssuch$
- **Agent State PDA (Optional but recommended for granular state)**:
  - **Seeds**: '["agent$_{s}tate$", $agent_{o}wner_{w}allet.key()$, $mint_{a}ccount.key()$, $program_{i}d$]'**Purpose** : $IftheKYAprogr$
    $chainstatespecifictoaparticularagentthatisn'tsuitablefornFTmetadata(e.g., a simple"active/inactive"fl$
    $chainauthorizedactions), adedicatedPDAperagentcanbeused.Thisavoidsfrequentandmoreexpensive NFT$

- `System Account (Payer):` The `Pubkey` of the account that signs the transaction and pays for the associated transaction fees (rent for new accounts and compute units for execution). This will typically be the KYA platform's operational wallet during initial registration, or the user's wallet for subsequent updates they initiate.

2.2.3 Solana Program Instructions (Functions)

The `kya_program` will expose a set of instructions (functions) that can be invoked via Solana transactions. These instructions define the core logic and interactions within the KYA system. Each instruction takes a set of accounts and arguments.

- `create_kya_agent_nft` **Instruction:**
  - **Purpose**: Mints a brand new KYA NFT, effectively registering a new AI agent's unique identity on the Solana blockchain.

- **Accounts Required (Input)**:
  - `payer`: `Pubkey` of the account paying for rent and transaction fees (writable, signer).
  - `mint_account`: `Pubkey` of the new `KYA_NFT_Mint_Account` to be created (writable).
  - `token_account`: `Pubkey` of the new `KYA_NFT_Token_Account` to be created to hold the NFT (writable).
  - `agent_owner_wallet`: `Pubkey` of the AI agent's ultimate owner (e.g., individual user's wallet, enterprise treasury wallet) (signer). This account will own the minted NFT.
  - `rent_sysvar`: Solana's `rent` system variable, required for account creation.
  - `token_program`: The SPL Token Program ID, as the `kya_program` invokes its instructions.
  - `system_program`: Solana System Program ID, for creating new accounts.
  - `token_metadata_program`: The Metaplex Token Metadata Program ID, used for attaching metadata to the NFT.
  - `metadata_account`: `Pubkey` of the new Metaplex metadata account to be created for the NFT (writable).
- **Data (Instruction Arguments)**:
  - `agent_name`: `String` (e.g., "FoodieBot"). This will be the NFT's public name.
  - `github_repo_url`: `String` URL to the agent's unique GitHub KYA repository (e.g., `https://github.com/myuser/foodiebot-kya`).
  - `initial_commit_hash`: `String` (SHA-1 or SHA-256 hash of the initial commit of the GitHub repo). This is crucial for verifying the integrity of the off-chain data.
  - `agent_type`: `u8` enum (e.g., `0` for individual, `1` for enterprise).
- **Logic (Simplified Flow)**:
  (i) Invoke `SystemProgram::create_account` to allocate space and fund the `mint_account` and `token_account`.
  (ii) Invoke `spl_token::instruction::initialize_mint` to set up the `mint_account` as an NFT (supply 1, decimals 0).
  (iii) Invoke `spl_token::instruction::mint_to` to mint the single token of this new NFT into the `token_account` owned by `agent_owner_wallet`.
  (iv) Invoke `spl_token::instruction::set_authority` to remove (or transfer) the 'mint$_{a}$*uthority*'*fromthein*
  mintable.Invoke`token_metadata_program::create_metadata_account`(*Metaplex*)*tocreateandpopulat*

(v) **`update_kya_metadata` Instruction:**

- **Purpose**: Allows the current agent owner to update specific fields in the NFT's metadata, primarily the `github_repo_url` or the `commit_hash`, when the off-chain GitHub profile changes. This ensures the on-chain pointer remains current.
- **Accounts Required (Input)**:
  - `agent_owner_wallet`: `Pubkey` of the AI agent's current owner (signer, writable). This is the authorization gate; only the rightful owner can initiate updates.
  - `mint_account`: `Pubkey` of the `KYA_NFT_Mint_Account` (writable).
  - `metadata_account`: `Pubkey` of the Metaplex (or custom) metadata account associated with the NFT (writable). This PDA is derived from the `mint_account`.

    * `token_metadata_program`: The Metaplex Token Metadata Program ID.
  - **Data (Instruction Arguments)**:
    * `new_github_repo_url`: `String` (optional, for changing the base repo URL, though this should be rare).
    * `new_commit_hash`: `String` (required when the content of the GitHub repo changes, pointing to the latest verified commit).
    * `new_agent_type`: `u8` enum (optional, if an agent's categorization needs to be updated).
  - **Logic**:
    (i) Verify that `agent_owner_wallet` is indeed the current owner of the `mint_account` by inspecting the associated `token_account`.
    (ii) Invoke `token_metadata_program::update_metadata_account` (Metaplex) to update the relevant fields within the `metadata_account` with the new URL, hash, and/or type. This update's validity is inherently tied to the `agent_owner_wallet`'s digital signature on the transaction.

- **`verify_agent_identity` (Off-Chain RPC Query / On-Chain Helper):**
  - **Purpose**: While the core identity verification happens off-chain (fetching GitHub data, checking digital signatures), the `kya_program` provides the essential on-chain reference. Platforms will primarily use Solana RPC calls to query the NFT's metadata directly. A simpler on-chain helper could exist for other smart contracts to query.
  - **Accounts Required (Input)**: `mint_account`: `Pubkey` of the `KYA_NFT_Mint_Account`.
  - **Data (Instruction Arguments)**: (None for a simple query).
  - **Logic**:
    (i) Read the `mint_account` data and its associated metadata account.
    (ii) Return the `github_repo_url` and `commit_hash` (and `agent_type`) from the metadata.
  - **Note on Digital Signature Verification**: The actual digital signature verification (where an agent signs a message and the verifier checks it against the public key derived from the NFT owner's wallet) is typically handled either off-chain by the integrating platform or through a separate, generic Solana 'ed25519' signature verification instruction provided by Solana itself, not directly within the `kya_program`'s custom logic. The `kya_program` merely stores the immutable reference to the off-chain data.

- **`transfer_kya_agent_nft` (Leveraging SPL Token Program):**
  - **Purpose**: Allows the ownership of a KYA NFT (and thus the agent's identity and control) to be transferred to a new owner. This leverages the standard SPL Token program's 'transfer' instruction directly, rather than requiring custom logic within the `kya_program`.
  - **Accounts Required (Input)**: Standard SPL `transfer` accounts (`source_token_account`, `destination_token_account`, `owner` (signer of transfer), `mint`, `token_program`).
  - **Logic**: Standard SPL Token transfer logic. The KYA program itself doesn't need custom logic here, as long as the NFT's 'transfer' authority isn't frozen or restricted by a specific '$\text{freeze}_{a}uthority$'.$This highlights the interoperability with existing Solana standards.$

2.2.4 Metadata Standards (Metaplex Integration)

The KYA system will fully utilize and extend the **Metaplex Token Metadata Standard** for storing the agent's on-chain metadata. This provides a widely adopted, interoperable, and standardized way to attach rich attributes to the NFT.

- **Metaplex Metadata Account**: A Program Derived Address (PDA) derived from the NFT's mint account and the Metaplex program ID. This account holds a JSON URI (often IPFS) and custom attributes.
- **Custom Attributes for KYA**: While Metaplex usually links to an external JSON URI for detailed metadata, for core, immutable KYA data, directly embedding the `GitHub Commit Hash` and `GitHub Repo URL` as 'attributes' within the on-chain metadata account itself offers greater immutability and direct verifiability without needing an extra IPFS lookup for these critical pointers. The full, detailed JSON profile would still reside on GitHub, referenced by this URL and hash.

```
{
  "name": "FoodieBot",
  "symbol": "KYA",
  "description": "Know Your Agent ID for FoodieBot, owned by [Owner Name/ID].",
  "image": "https://kya.org/assets/foodiebot.png", // Optional agent avatar/logo
  "properties": {
    "files": [
      {"uri": "https://github.com/user/foodiebot-kya-profile/commit/<commit_hash>", "
    ]
  },
  "attributes": [
    {"trait_type": "Agent Type", "value": "Individual"},
    {"trait_type": "KYA Schema Version", "value": "1.0.0"}, // Current schema version
    {"trait_type": "GitHub Commit Hash", "value": "<latest_verified_commit_hash>"},
    {"trait_type": "GitHub Repo URL", "value": "https://github.com/user/foodiebot-kya
  ]
}
```

This detailed smart contract architecture ensures that the KYA system leverages Solana's high performance and robust account model to provide a secure, scalable, and verifiable identity layer for AI agents, while maintaining the flexibility and richness of off-chain data managed via GitHub.

3 Benefits of the KYA System

The KYA system offers transformative advantages for the burgeoning AI agent ecosystem, addressing fundamental needs for trust, security, and interoperability that are not adequately met by current identity solutions.

- **Enhanced Security and Trust**: By providing a standardized, cryptographically verifiable identity for AI agents, the KYA system drastically reduces the risk of fraud, impersonation, and unauthorized access. The immutability of the blockchain record for the KYA NFT and the auditable, version-controlled nature of Git for the KYA profile ensure transparency and accountability at every step of an agent's lifecycle. Platforms can trust that an agent is who it claims to be and is authorized for its actions.
- **Streamlined Commerce and Interactions**: Seamless authentication and authorization mechanisms facilitate smooth and efficient interactions between AI agents and diverse platforms. This reduces friction in transactions, data exchange, and service consumption, leading to a more dynamic and accessible agentic economy. Agents can participate in e-commerce, access financial services, and integrate with enterprise systems with unprecedented ease and security.
- **Granular Permissioning and Control**: The detailed, extensible schema within the GitHub repository allows for highly granular control over agent capabilities, spending limits, data access, and authorized platforms. This empowers individual users and large organizations to define precise boundaries for their agents' autonomous actions, mitigating risks associated with over-permissioned or rogue AI.
- **Transparency and Auditability**: The use of GitHub for KYA data provides a transparent and auditable record of an agent's identity history and permissions. Every change to an agent's profile, including modifications to its capabilities or permissions, is tracked via Git commits and referenced by an immutable hash on the blockchain. This fosters greater trust and enables forensic analysis if disputes or issues arise.
- **Interoperability**: A standardized KYA framework promotes true interoperability across different AI models, development platforms (e.g., Google Gemini, OpenAI, Microsoft Azure), and service providers. Agents created using various AI frameworks can seamlessly interact with a wide range of applications and services that integrate with the KYA system, unlocking a truly connected agent ecosystem.
- **Decentralization and User Control**: By anchoring agent identities on a decentralized blockchain (Solana) and allowing users to manage their agent's KYA data in their own GitHub forks, the system empowers individuals and organizations with greater autonomy and control over their AI agents. This minimizes reliance on central authorities for identity validation, providing censorship resistance and user sovereignty.
- **Reduced Credential Sprawl**: By centralizing agent identity and linking to secure credential management, the KYA system minimizes the need for agents to directly hold or manage numerous API keys or passwords, simplifying security posture for both agent owners and integrating platforms.

4 Use Cases and Enterprise Integration

The KYA system's versatility allows for a wide array of applications, providing significant value for both individual users and large enterprises.

4.1 Individual User Example: Ordering Food on DoorDash

This scenario demonstrates how a personal AI agent can securely and autonomously engage in e-commerce using the KYA system.

1. **Agent Creation & KYA Registration**:

- A user creates a personal AI agent (e.g., "FoodieBot") using an AI development platform like Google Gemini, OpenAI's API, or Microsoft Azure AI Studio.
- The user then brings this nascent agent to the KYA platform for identity registration.

2. **KYA Onboarding**:

- During a guided onboarding process on the KYA platform's user interface, the system automatically forks a base GitHub schema, creating a unique, dedicated repository for "FoodieBot" (e.g., https://github.com/your-username/foodiebot-kya-profile).
- The user populates the essential KYA data within this GitHub repository via the KYA UI:
  - `agent.json`: Agent name ("FoodieBot"), type ("individual"), purpose ("Automates food ordering").
  - `data/capabilities.json`: Specifies functionalities like "can make purchases", "understands natural language food orders".
  - `data/authorized_actions.json`: Explicitly lists "order food on DoorDash", "process payments via authorized methods".
  - `data/spend_limits.json`: Sets financial guardrails, e.g., `{"daily_limit": $50, "transaction_limit": $30, "authorized_categories": ["food"]}`.

3. **Secure Payment Profile Creation**:

- The user securely provides payment details (e.g., credit card information) via the dedicated KYA mobile or web application. This information is **encrypted** at the client-side.
- The encrypted payment data (or a tokenized reference from a payment gateway) is stored either locally on the user's device (for enhanced privacy and control, e.g., in a secure enclave) or within a secure, encrypted vault managed by the KYA application's backend. A secure reference or identifier to this vault is stored in `data/payment_profiles.json` in the agent's GitHub repo. The raw payment details are never exposed to the GitHub repository.

4. **NFT Minting and Private Key Generation**:

- The KYA platform then mints a unique Solana NFT for "FoodieBot" using the `create_kya_agent_nft` instruction on the `kya_program`.
- The NFT's metadata contains the URL and initial commit hash pointing to "FoodieBot's" GitHub repository.
- Crucially, a unique **private cryptographic key** is generated for the user's control over this agent's identity. This key is stored securely (e.g., directly in a hardware wallet, a secure enclave on the user's smartphone, or encrypted within the KYA app's secure storage). This key is essential for signing transactions and authorizing the agent's actions.

5. **DoorDash Interaction and Verification Flow**:

- When "FoodieBot" attempts to order food on DoorDash (e.g., triggered by a user voice command or a recurring schedule):
  (a) **Identity Presentation**: "FoodieBot" (or its managing software) presents its Solana NFT ID to DoorDash.
  (b) **Signature Generation**: The agent's managing system (e.g., the KYA mobile app or a secure backend service controlled by the user) uses the agent's **private cryptographic key** to generate a **digital signature** over a challenge or the transaction details provided by DoorDash. This signature proves that the entity controlling "FoodieBot" is the legitimate owner of the NFT.
  (c) **On-Chain Verification**: DoorDash's system (or an integrated KYA verification module) queries the Solana blockchain to verify the NFT's authenticity and validates the received digital signature against the public key associated with the NFT's owner. This confirms ownership and legitimacy.
  (d) **GitHub Data Retrieval**: Upon successful on-chain verification, DoorDash retrieves the specific **commit hash** and GitHub repository URL from the NFT's metadata. It then accesses "FoodieBot's" GitHub repository to fetch the KYA profile data at that precise commit.
  (e) **Permission Check**: DoorDash's system reads `data/authorized_actions.json` to confirm "FoodieBot" is authorized to "order food on DoorDash." It also checks `data/spend_limits.json` against the current order total and daily limits.
  (f) **Secure Payment Initiation**: If all permissions and limits are satisfied, DoorDash initiates a secure payment request, referencing the identifier from "FoodieBot's" `data/payment_profiles.json`.
  (g) **Ephemeral Credential Use**: The KYA app on the user's device (or the secure vault) receives this request, decrypts the actual credit card details using the user's private decryption key (which is never exposed), and securely transmits them to DoorDash's payment gateway via a transient, encrypted channel. The raw credit card details are never exposed to the agent itself or persistently stored by DoorDash.

Result: The food order is successfully placed, with full auditability and verified authorization, demonstrating seamless and secure AI agent participation in commerce.

4.2 Enterprise Client Example: Sales Agent Accessing CRM Data (using AD Groups)

This illustrates how an enterprise AI agent can securely access sensitive corporate data, with permissions managed through existing Active Directory (AD) groups, highlighting KYA's interoperability with enterprise IAM.

1. **Base Schema for Enterprise Agents**: The KYA platform provides a specialized base schema for enterprise agents, incorporating fields relevant to organizational structures, compliance requirements, and internal system access policies. 2. **Agent Creation & KYA Onboarding**:

- An enterprise creates an AI agent (e.g., "SalesAssistBot") designed to automate tasks for its sales department.
- During KYA onboarding, the enterprise IT department (or an authorized admin) forks the base enterprise schema, creating "SalesAssistBot's" dedicated GitHub repository (e.g., https://github.com/mycorp/salesassistbot-kya-profile).

3. **Active Directory (AD) Integration**:
   - The KYA platform is pre-integrated or configured to communicate with the enterprise's existing Active Directory (AD).
   - When "SalesAssistBot" is registered, its `agent.json` is configured to specify its `owner_id` as an internal employee ID or a specific service account that is directly linked to an AD Group (e.g., "CRM-Access-Sales", "Customer-Data-Read").
   - The enterprise's KYA schema (within its GitHub fork) defines data access permissions (e.g., "read-only_customer_contact_info," "write_new_leads," "access_sales_pipeline") that are dynamically mapped to corresponding AD Group policies. This linkage is a configuration, not a hard-coded integration for every system.
   - The `data/authorized_platforms.json` for "SalesAssistBot" explicitly lists "Salesforce CRM" as an authorized platform.
4. **NFT Minting**:
   - A Solana NFT is minted for "SalesAssistBot" using the `create_kya_agent_nft` instruction, pointing to its GitHub repository.
   - The private key for this NFT is securely managed by the enterprise's IT infrastructure, typically within a Hardware Security Module (HSM) or a robust Key Management System (KMS), providing centralized and highly secure control.
5. **Salesforce Interaction**:
   - "SalesAssistBot" attempts to retrieve customer contact information from Salesforce (e.g., in response to a sales rep's query or as part of a lead nurturing workflow):
     (a) **Identity Presentation**: "SalesAssistBot" presents its Solana NFT ID to Salesforce's API gateway.
     (b) **Signature Generation**: The enterprise's secure backend (e.g., a service integrated with the KMS/HSM) uses "SalesAssistBot's" private key to generate a digital signature over the Salesforce API request.
     (c) **On-Chain Verification**: Salesforce's KYA integration module queries the Solana blockchain to verify the NFT's authenticity and validates the digital signature.
     (d) **GitHub Data Retrieval**: Upon successful verification, Salesforce retrieves the commit hash and URL from the NFT's metadata and accesses "SalesAssistBot's" GitHub repo.
     (e) **Permission Check (with AD Integration)**: Salesforce's KYA module reads `data/authorized_actions.json` (e.g., confirming "read_customer_contact_info"). Crucially, it then uses the `owner_id` from `agent.json` to query the enterprise's Active Directory (via a secure, internal API or federated identity solution) to confirm "SalesAssistBot's" association with the "CRM-Access-Sales" AD Group.

(f) **Dynamic Authorization**: Based on the verified AD group membership, which is mapped to the KYA schema's defined data access rules (e.g., "read-only access to customer contact details" within the enterprise's configured KYA policies), Salesforce grants or denies the request.

Result: "SalesAssistBot" successfully retrieves authorized customer data, with its actions fully audited and its permissions dynamically enforced based on the enterprise's existing AD infrastructure and the KYA schema. This minimizes manual permission configuration for each agent and leverages existing enterprise security policies.

5 Comparison to Existing Technologies

Traditional identity and access management (IAM) solutions, while effective for human users and conventional applications, fall short in addressing the unique requirements of autonomous AI agents. This section compares our KYA system to established identity protocols like OAuth 2.0 and OpenID Connect (OIDC), highlighting the fundamental differences and the novel advantages our approach brings to the agentic AI landscape.

5.1 Limitations of Traditional IAM for AI Agents

Existing IAM frameworks, including OAuth and OIDC, were primarily designed for:

- **Human-centric interactions**: They assume a human user in the loop for authentication (e.g., login screens, MFA prompts, consent dialogues). AI agents lack a direct "login" interface and require machine-to-machine authentication.
- **Static, pre-provisioned access**: Permissions are often broad and tied to roles, not dynamic capabilities or granular, context-aware actions. An AI agent's required permissions can change frequently based on its evolving tasks.
- **Long-lived sessions**: Designed for human user sessions that might last hours or days, not the potentially ephemeral, rapidly spinning up and down nature of many AI agents, where credentials need to be highly transient.
- **Centralized Trust**: Reliance on a single Identity Provider (IdP) for authentication introduces a single point of failure or control, which can be a bottleneck or a target for attack in a highly distributed AI agent ecosystem.
- **Limited Auditability of Agent Intent**: While human actions are logged, tracing the *delegation chain*, *specific capabilities*, and *intent* behind an AI agent's autonomous, context-driven actions is difficult.
- **Credential Sprawl**: AI agents often resort to API keys or service accounts, leading to unmanaged credentials, over-permissioning (granting more access than needed), and significant audit gaps.
- **Lack of AI-specific context**: Traditional systems don't natively understand AI agent types, their capabilities, or their behavioral constraints.

5.2 Comparison with OAuth 2.0

OAuth 2.0 is an authorization framework for delegated access, allowing an application (client) to access resources on behalf of a user.

Feature	**OAuth 2.0 (Traditional for Apps)**	**KYA System (AI Agents)**
Primary Purpose	Authorization (delegating user's access to client app)	Identity and Authorization for the AI Agent itself
Core Subject	Human user (whose permission is delegated)	The AI Agent as an autonomous entity
Identity of Client (Agent)	Opaque client ID/secret; generic	Rich, verifiable identity (name, type, purpose, capabilities)
Verification Model	Centralized Authorization Server	Decentralized (Solana NFT ownership via signature)
Permission Detail	Scopes (broad strings); often static	Granular, version-controlled JSON schema (actions, limits, platforms, data access)
Credential Mgmt.	Client credentials (shared); bearer tokens	Encrypted credentials (stored off-chain), ephemeral decryption
Transparency	Opaque access tokens; internal IdP logs	Publicly auditable GitHub repo (KYA profile) and blockchain records
Version Control	Not inherent to protocol; relies on API versioning	Built-in versioning of KYA profile via Git commits
Delegation Depth	User → App → Resource	Owner → Agent → Resource; agent-specific permissions
Flexibility for AI	Retro-fitted (e.g., Client Credentials Flow)	Native, designed for complex, dynamic AI agent behaviors

5.3 Comparison with OpenID Connect (OIDC)

OpenID Connect is an authentication layer built on top of OAuth 2.0, allowing clients to verify the identity of an end-user based on authentication performed by an Authorization Server.

Feature	**OpenID Connect (Traditional for User Auth)**	**KYA System (AI Agents)**
Primary Purpose	User authentication + authorization	AI Agent authentication + authorization
Core Subject	Human end-user	The AI Agent itself
Identity Token	ID Token (JWT with user claims)	Solana NFT (immutable pointer to agent profile)

Table 2: Comparison of KYA with Traditional IAM (Continued)

Feature	**OpenID Connect (Traditional for User Auth)**	**KYA System (AI Agents)**
Identity Verification	Centralized IdP	Decentralized (Solana blockchain)
Agent Profile Data	Basic claims in ID Token/UserInfo	Rich, version-controlled GitHub profile (agent-specific)
Lifecycle Mgmt.	Human user session lifecycle	AI Agent lifecycle (creation, updates, deactivation)
Auditability	User login/action logs	Agent action logs + Git commit history of KYA profile
Interoperability	Standardized for human logins	Standardized for AI agent interactions

5.4 Novelty and Improvements of the KYA System

Our KYA system introduces several key innovations that address the unique challenges of AI agent identity, moving beyond the limitations of existing solutions:

- **Native AI Agent Identity:** Unlike systems that retro-fit human identity paradigms to agents, KYA establishes a first-class, verifiable identity for the AI agent itself. The agent, rather than merely acting *on behalf of* a human or being a generic client, possesses its own digital passport tied to its unique capabilities and purpose.
- **Decentralized and Verifiable Ownership:** The use of Solana NFTs and strong cryptographic signatures provides a decentralized, immutable, and publicly verifiable proof of an agent's ownership and control. This significantly reduces reliance on a single centralized authority for core identity attestation, a major improvement over traditional IdP-dependent systems.
- **Transparent and Auditable KYA Profile:** The GitHub repository as the granular source for the agent's KYA data offers unparalleled transparency and auditability. Every change to an agent's capabilities, permissions, or associated metadata is tracked via Git's inherent version control, creating a fully auditable history accessible to authorized parties. This level of detailed, auditable information is not natively available or standardized in OAuth/OIDC.
- **Granular, Dynamic, and Version-Controlled Permissions:** Our detailed, JSON Schema-based KYA profile allows for highly specific and dynamic definition of an agent's authorized actions, spend limits, platform access, and data interaction rules. This goes beyond static scopes by enabling adaptive authorization that can evolve with the agent's tasks and context, with clear versioning for future changes and immediate impact.
- **Clear Separation of Public Profile and Private Credentials:** The KYA system meticulously distinguishes between the publicly verifiable identity/profile (via NFT and GitHub) and securely stored, encrypted operational credentials ('credentials/' directory). This minimizes the attack surface while enabling secure, just-in-time credential usage when needed, without exposing raw secrets to the agent or public.

- **"Trust by Inspection"**: By providing a publicly verifiable NFT (with its embedded commit hash) and a link to a version-controlled KYA profile on GitHub, platforms can "inspect" an agent's authorized capabilities, history, and defined limitations *before* granting access or processing a request, fostering greater inherent trust in the agent ecosystem.
- **Interoperability and Standardization for Agents:** By establishing a common framework and schema for AI agent identity, the KYA system promotes true interoperability across different AI models, development platforms, and service providers. This is a critical need that current ad-hoc, proprietary solutions fail to meet, leading to fragmented ecosystems.

In essence, while OAuth and OIDC are foundational for delegated human access, our KYA system provides the missing **decentralized, verifiable, and richly descriptive identity layer specifically for AI agents**, enabling a new paradigm of secure, accountable, and highly functional agentic interactions in the digital economy. This distinct focus on the AI agent as a primary identity subject, coupled with the novel combination of blockchain immutability and Git's version control for dynamic KYA profiles, forms the core of our patentable innovation.

6 Schema Versioning and Evolution

The dynamism of AI agent capabilities, the rapid pace of AI development, and evolving regulatory landscapes necessitate a flexible yet robust approach to schema evolution. Our KYA system is designed to accommodate changes to the underlying identity schema without compromising existing agent functionality or requiring widespread, disruptive updates. This section outlines the strategies for managing and propagating these schema changes.

6.1 Semantic Versioning for Schemas

The KYA base schema (the foundational structure that individual and enterprise agents fork) will strictly adhere to **semantic versioning** (MAJOR.MINOR.PATCH), a widely adopted standard for software versioning. This provides clear guidelines for backward compatibility and the severity of changes.

- **MAJOR Version (e.g., 1.0.0 to 2.0.0):**
  - **Indicates Backward-Incompatible Changes**: These are significant alterations that might remove required fields, change data types in a non-nullable way, fundamentally restructure core components, or introduce breaking validation rules.
  - **Action Required**: Such updates will necessitate explicit action from agent creators/owners to migrate their agent's KYA data to the new schema. Automated migration tools will be provided by the KYA platform, but human review and confirmation may be required for complex changes.
- **MINOR Version (e.g., 1.0.0 to 1.1.0):**
  - **Signifies Backward-Compatible, Additive Changes**: This includes adding new optional fields, new data categories (e.g., a new type of `data/` subdirectory), or new enumeration values to existing fields. Existing required fields are not altered or removed.

  - **Forward Compatibility by Design**: Agents operating on older minor versions will still function correctly without immediate updates. Platforms are designed to gracefully handle and ignore any new, unrecognized fields from a newer schema version if the agent hasn't updated its own profile yet.
- **PATCH Version (e.g., 1.0.0 to 1.0.1):**
  - **Represents Backward-Compatible Bug Fixes or Minor Improvements**: These are typically non-functional corrections to the schema definitions (e.g., clarifying descriptions, fixing validation errors in the schema itself), or very minor additive changes that have no impact on existing data. No explicit action is required from agent owners for these updates.

6.2 Base Schema Management

The core KYA schema will reside in a publicly accessible, version-controlled GitHub repository managed by the KYA system's maintainers. This "base repo" will serve as the single source of truth for all KYA schema definitions.

- **Formal Change Process:** All proposed changes to the base KYA schema will undergo a formal review process. This process will involve community feedback, technical review by KYA maintainers, and potentially a governance vote (e.g., through a DAO for significant updates) to ensure changes are well-considered, secure, and aligned with the ecosystem's needs.
- **Release Cadence:** New versions of the base schema will be released on a defined cadence (e.g., quarterly for minor versions, annually for major versions as needed). Each release will be accompanied by comprehensive release notes detailing all changes, providing clear migration guides (for major versions), and outlining backward compatibility considerations.

6.3 Agent-Specific Schema Forks and Updates

Each AI agent's KYA profile is an independent *fork* of the base schema on GitHub. This distributed model provides significant advantages for managing evolution and empowers agent owners.

- **Independent Updates:** Agent owners maintain full control over their specific GitHub fork. They can choose precisely when to update their agent's KYA schema to a newer version of the base schema, allowing for testing and phased rollouts within their own operational environments.
- **Forward Compatibility by Design (for MINOR/PATCH):** As noted, for MINOR and PATCH updates, agent's KYA profiles (which are instances of older schema versions) will automatically remain "forward-compatible." When platforms query an agent's KYA data, their parsing logic is designed to gracefully handle and ignore any new, unrecognized fields from a newer schema version if the agent hasn't yet updated its own profile. This prevents breaking changes for systems that haven't synchronized their schema version.

- **Backward Compatibility through Dual-Reading/Transformation (Platform Responsibility):** Platforms integrating with the KYA system will be designed to support multiple recent versions of the KYA schema (e.g., the current major version and the immediate prior major version, along with all their minor/patch versions). This is particularly important during transition periods for major version changes.
  - **Schema Adapters**: Platforms can implement internal data transformation layers or "schema adapters" to normalize incoming KYA data from various schema versions to their internal, standardized data model.
  - **Major Version Deprecation**: In cases where a platform only supports a newer major version, agents operating on older major versions would be flagged by the platform for an update.

6.4 Graceful Handling of Schema Changes by Agents

AI agents themselves, or more accurately, the systems managing them on behalf of their owners, will be equipped with mechanisms to handle schema evolution gracefully and to facilitate necessary updates.

- **Discovery of New Versions:** The KYA system's core Solana smart contract (or an associated off-chain oracle service) can broadcast or provide a public API to query the latest available base KYA schema version. This allows agent management systems to proactively check for updates.
- **Agent-Side Validation and Notification:**
  - When a new MAJOR version of the base KYA schema is released, the KYA platform will automatically detect if an agent's profile (based on its on-chain commit hash and GitHub repo) is on an older major version.
  - Agent owners will be notified via registered contact methods (email, in-app notifications) about the new version and the requirement to migrate their agent's KYA profile.
  - Agent management interfaces (e.g., the KYA web UI for individuals, enterprise KYA management dashboards) will guide owners through the migration process. This might involve prompting them to update their GitHub fork (pulling changes from the base repo), review new required fields, and populate any new necessary data.
- **Automated and Assisted Migration Tools:** The KYA platform will provide tools to assist in the migration process. These could:
  - Automatically update the agent's GitHub fork, adding new optional fields with default values where applicable.
  - Highlight fields that require manual input from the agent owner due to breaking changes or new essential data requirements.
  - Validate the updated schema against the new version rules before committing and updating the Solana NFT.
- **"Grace Period" for Major Version Migrations:** For backward-incompatible (MAJOR) schema changes, a defined "grace period" will be established (e.g., 3-6 months). During this

period, platforms may continue to support the previous major version, but potentially with warnings or degraded functionality for non-migrated agents. This provides ample time for owners to update their agents. After the grace period, support for the deprecated major version might cease, making agent functionality dependent on migration.

- **Rollback Capability (Git's Strength):** A key advantage of using GitHub for KYA data is the inherent version control. If an update to an agent's KYA profile causes unforeseen issues or misconfigurations, the owner can easily revert their GitHub fork to a previous, stable commit. The associated Solana NFT's metadata can then be updated via the `update_kya_metadata` instruction to point to this reverted commit hash, effectively rolling back the agent's identity state.

6.5 Solana Smart Contract Upgradeability

While the KYA data resides off-chain in GitHub, the Solana smart contracts that manage the NFT and verification pointers on-chain may also need updates (e.g., bug fixes, new features, security enhancements). Solana programs are **upgradable by default**, which is a significant advantage for long-term system maintenance.

- **Upgrade Authority:** Each KYA smart contract deployed on Solana will have an `upgrade_authority`. This authority must be securely managed (e.g., via a multi-signature wallet for the KYA system maintainers, a dedicated governance DAO, or a time-locked treasury). The `upgrade_authority` is the only key capable of pushing new program code.
- **Proxy Pattern (Optional but Recommended for Complex Upgrades):** For more robust and transparent upgrades, a proxy pattern can be employed. This involves a fixed "proxy" contract that maintains a pointer to the current "logic" contract. When an upgrade occurs, only the pointer in the proxy contract is changed to point to the new logic contract. This allows the core program logic to be swapped out without changing the KYA NFT's associated program ID, making upgrades transparent to existing agents and integrating platforms and avoiding the need to re-mint NFTs.
- **Transparency and Notification:** All smart contract upgrades will be publicly announced by the KYA system maintainers and are inherently verifiable on the Solana blockchain (as the program ID changes or the proxy pointer updates). Users and integrating platforms can monitor these upgrades and react accordingly.
- **Backward Compatibility of On-Chain State**: Smart contract upgrades must be carefully designed to be backward compatible with existing on-chain data structures, ensuring that previously created NFTs and their associated data remain valid and accessible.

By combining the robust versioning capabilities of Git for KYA data with the secure and adaptable upgradability of Solana smart contracts, the KYA system ensures a resilient and adaptable framework for AI agent identity in an ever-evolving digital landscape.

7 Key Management and Security Protocols

The security of the KYA system hinges upon the integrity and confidentiality of the cryptographic keys used for agent identification, authentication, and secure access to sensitive data. Compromise of these keys would undermine the entire trust model. This section elaborates on the secure generation, storage, and usage of these private keys, differentiating strategies for individual users and enterprise clients, and outlining rigorous protocols for secure credential access.

7.1 Private Key Generation

The private key associated with each AI agent's Solana NFT is the fundamental component for proving ownership and authorizing actions. Its secure generation is the first critical step.

- **Cryptographically Secure Randomness:** All keys must be generated using cryptographically strong pseudo-random number generators (CSPRNGs) with sufficient entropy. For maximum security, hardware-based entropy sources (e.g., within dedicated Hardware Security Modules - HSMs, or Trusted Execution Environments - TEEs) are strongly recommended, especially for high-value enterprise agents or for the master keys from which agent keys are derived.
- **Determinism for Recovery (Optional for Enterprise):** For enterprise clients managing numerous agents, a hierarchical deterministic (HD) key derivation scheme (analogous to BIP32/BIP39 standards used in cryptocurrency wallets) could be considered. This allows a single master seed key to deterministically derive multiple agent-specific keys. This simplifies backup and recovery procedures for large fleets of agents while maintaining individual agent security, as compromise of one derived key does not compromise the master key if proper key derivation hardening is applied.
- **User-Controlled vs. System-Controlled Generation:**
  - **Individual Users:** The private key can be generated directly on the user's secure device (e.g., smartphone), ensuring the key never leaves the device.
  - **Enterprises:** Keys might be generated within an organizational KMS or HSM, maintaining central control and auditing over the key generation process.

7.2 Private Key Storage

The storage of private keys is paramount to prevent unauthorized access and compromise. Different approaches are necessary for individual and enterprise contexts due to varying security requirements, compliance needs, and infrastructure.

- **For Individual Users:**
  - **User-Controlled Wallets:** Private keys will ideally reside in the user's personal, secure wallet solution. The KYA mobile/web app will facilitate interaction with these:
    - **Hardware Wallets (HSMs/Secure Elements):** The gold standard for cold storage. The private key never leaves the hardware device. All signing operations

occur securely within the device. The KYA app interfaces with the hardware wallet via standard protocols (e.g., WebHID, Bluetooth).
    - **Secure Enclaves (e.g., Apple Secure Enclave, Android Keystore):** Built-in hardware security features on modern smartphones that provide a secure, isolated environment for cryptographic operations. Keys are generated and stored within the enclave, protected from the main operating system and other applications. The KYA mobile app would leverage these.
    - **Browser-based Wallets (with caution):** For convenience, hot wallets in browsers might be supported, but users will be strongly advised about the inherent risks of online exposure. Keys would be encrypted and password-protected locally, with frequent re-authentication.
  - **Encrypted Storage on Device (for decryption keys):** For sensitive credentials that are referenced in the agent's `credentials/` directory on GitHub, the actual decryption key to unlock these credentials would reside within the KYA app's secure storage (e.g., Secure Enclave, or encrypted using a robust user-derived passphrase) on the user's device. This ensures the private credential itself is never stored in plain text or publicly accessible, and its decryption is user-controlled.
- **For Enterprise Clients:**
  - **Hardware Security Modules (HSMs):** Enterprises managing high-value agents or large fleets will utilize FIPS 140-2/3 certified HSMs. HSMs are tamper-resistant physical devices designed specifically for secure cryptographic key generation, storage, and management. Keys never leave the HSM for signing operations.
    - **On-Premises HSMs:** For maximum control and compliance in highly regulated industries.
    - **Cloud HSM Services (e.g., AWS CloudHSM, Azure Dedicated HSM):** Provide HSM functionality as a managed service, balancing strong security with scalability and operational ease.
  - **Key Management Systems (KMS):** Cloud-based KMS solutions (e.g., AWS KMS, Google Cloud KMS, Azure Key Vault) provide centralized, secure management of cryptographic keys, including key rotation, fine-grained access control (IAM policies), and comprehensive auditing. These can integrate with underlying HSMs for the highest level of root-of-trust.
  - **Multi-Party Computation (MPC) Wallets:** For collaborative agent management, enhanced security, or achieving threshold signatures, MPC allows a private key to be split into multiple shares. These shares are distributed among different parties or servers. A signature can be generated collaboratively (e.g., requiring 3 out of 5 shares) without any single party ever possessing the entire private key. This is ideal for scenarios requiring consensus among multiple organizational stakeholders for agent actions or for protecting against single points of compromise.
  - **Trusted Execution Environments (TEEs):** Technologies like Intel SGX or AMD SEV provide secure, isolated execution environments within a CPU. These can be used to protect code and data (including private keys) from the host operating system or

hypervisor. This could be used for agents whose runtime environment needs additional protection for key usage.

7.3 Private Key Usage and Digital Signatures

Private keys are primarily used for generating digital signatures to prove ownership of the Solana NFT and to authorize agent actions.

- **NFT Ownership Verification (Agent Authentication):**
  - When an AI agent initiates an interaction with a platform (e.g., a DoorDash order, a Salesforce API call), it presents its Solana NFT.
  - Concurrently, the agent (or more accurately, its managing system on behalf of the owner) uses its associated private key to create a **digital signature** over a specific challenge (e.g., a nonce) or the transaction payload provided by the platform. This signature is an asymmetric encryption of a hash of the data, verifiable by the corresponding public key.
  - This signature is then transmitted along with the NFT to the platform.
  - The platform verifies this signature on the Solana blockchain using the agent's public key (which is publicly derivable from the NFT's owner account). A valid signature cryptographically proves that the agent is indeed controlled by the legitimate owner of that specific NFT, preventing anyone from simply referencing an NFT without possessing its corresponding private key.
- **Authorization of Actions (Granular Permissions):**
  - For actions requiring explicit authorization (e.g., making a financial purchase, executing a data write operation, accessing sensitive data categories), the KYA system leverages the digital signature for approval.
  - The platform requesting the action would provide a clearly defined transaction payload or a signed request for the agent's owner to approve.
  - The agent's controlling entity (the user's KYA app with its secure enclave or the enterprise's KMS/HSM) would use the agent's private key to cryptographically sign this specific request.
  - The signed request is then submitted back to the platform, which verifies the signature to ensure the action is genuinely authorized by the agent's owner and aligns with the agent's defined KYA permissions. This provides non-repudiation for agent actions.

7.4 Secure Credential Access Protocols

Accessing sensitive credentials (like credit card numbers, login passwords, or API tokens) that are referenced in the 'credentials/' directory must adhere to strict security protocols to prevent leakage. The agent itself should never directly possess the raw sensitive data.

- **Zero-Knowledge Proofs (ZKPs) (Advanced Consideration):** In a highly advanced and privacy-focused implementation, ZKPs could be employed. An agent could cryptographically prove that it possesses knowledge of a specific, encrypted credential (e.g., a credit card

number) and that it's authorized to use it, *without ever revealing the raw credential itself* to the verifying platform. The ZKP would cryptographically confirm the agent's ability to fulfill the payment requirement without exposing the data. This significantly enhances privacy and reduces the data exposure risk.

- **Ephemeral Decryption and Secure Transmission:** This is the primary mechanism for secure credential use:
  - When an agent requires a specific credential (e.g., credit card details for a DoorDash order or an API token for a service), the interacting platform makes a secure request (e.g., via an encrypted API call) to the KYA application or the enterprise system that securely holds the agent's decryption key.
  - This KYA application/system, after verifying the authenticity of the platform's request and the agent's authorization (via its private key signature), performs the decryption of the credential *within its secure environment* (e.g., within the Secure Enclave, HSM, or KMS).
  - The decrypted credential is then transmitted **ephemerally** to the requesting platform *only for the duration of the transaction* over a secure, encrypted channel (e.g., TLS/SSL with mutual authentication). The raw credential is not persisted by the agent's ephemeral memory or the platform after use.
  - This process ensures that the AI agent itself never directly handles or "sees" the raw sensitive credential. It only orchestrates the request and handles the signed authorization, while the secure element handles the sensitive data.
- **Principle of Least Privilege (PoLP):** Agents are only granted access to the minimum necessary credentials required for a specific, authorized task. Permissions are fine-grained and contextual, dynamically adjusting based on the ‘authorized$_{a}ctions.json$‘$andotherKYAschemadefinitions.$**C**$blockchain-basedsecretstominimizetheimpactofpotentialcompromisesovertime.$
- **Audit Logging:** All key usage events, digital signature generations, decryption requests, and access attempts to credentials will be meticulously logged by the KYA platform and enterprise KMS/HSM for security auditing, compliance, and forensic analysis. This provides a clear trail of all sensitive operations.

By combining strong cryptographic primitives, robust key management practices tailored to different user segments, and a clear separation of identity from sensitive credentials, the KYA system ensures a highly secure and trustworthy environment for AI agent operations. This multi-layered approach to security, spanning key generation, storage, usage, and access protocols, is a cornerstone of the system's innovative and patentable aspects.

8 Error Handling and Edge Cases: Ensuring System Resilience

While the Solana blockchain provides inherent immutability and high availability for on-chain identity records, the KYA system's reliance on off-chain GitHub repositories for detailed agent profiles introduces potential points of failure that must be addressed with robust error handling and resilience strategies. This section outlines how the system is designed to maintain operation and data integrity under various adverse conditions.

8.1 GitHub Repository Related Edge Cases

The GitHub repository is the primary source for the comprehensive KYA profile. Its availability and integrity are paramount for an agent's continuous operation.

- **Scenario: GitHub Repository Deletion:**
  - **Detection**:
    - **Periodic Health Checks**: The KYA system will implement an off-chain monitoring service that periodically performs checks on the accessibility and existence of each agent's GitHub repository, referencing the URL stored in the Solana NFT metadata. This proactively identifies deleted or unreachable repositories.
    - **On-Demand Verification Failures**: When an integrating platform attempts to verify an agent's identity and its GitHub repository is unreachable (e.g., returns a 404 Not Found error), this event will be logged by the platform and reported to the KYA system's monitoring service.
  - **Handling**:
    - **Grace Period for Recovery**: GitHub typically allows repository restoration within 90 days. During this period, the KYA system will automatically alert the agent owner via all registered contact methods (email, in-app notification, API callback) about the missing repository and provide clear instructions for restoration.
    - **Status Flag on NFT**: The agent's Solana NFT metadata can be updated (via the `update_kya_metadata` instruction) to include a "status" flag (e.g., "active," "pending_repo_recovery," "deactivated"). A detected deleted repo would trigger a change to "pending_repo_recovery."
    - **Functionality Restriction**: Platforms integrating with KYA would be advised to implement policies to restrict or degrade functionality for agents with a "pending_repo_recovery" status, preventing unverified or unauthorized actions until the repository is restored.
    - **Permanent Deactivation**: After the restoration grace period, if the repository is not recovered, the KYA system may trigger a permanent "deactivated" status on the NFT. This effectively revokes the agent's verifiable identity until a new KYA registration (and NFT) is initiated by the owner.
    - **"Soft Delete" Option**: The KYA platform might offer a "soft delete" feature for agents, allowing the user to temporarily unpublish their KYA data from public verification (e.g., by changing the NFT status) while retaining the GitHub repo, offering flexibility without full deletion.
- **Scenario: GitHub Repository Corrupted or Tampered:**
  - **Detection**: The Solana NFT stores a specific **commit hash** of the GitHub repository. When a platform fetches the KYA data from GitHub, it will perform a cryptographic hash (e.g., SHA-256) of the retrieved content and verify that this computed hash matches the `commit_hash` recorded on the NFT. Any mismatch indicates that the off-chain data has been altered since its last on-chain verification.

  - **Handling**:
    * **Verification Failure**: If the fetched content's hash does not match the NFT's recorded hash, the identity verification process for that agent immediately fails, and the platform will deny the agent's requested action.
    * **Alerting Owner**: The KYA system's monitoring service would detect such a mismatch during its routine integrity checks and alert the agent's owner, prompting them to investigate the integrity of their GitHub repository and restore a valid version.
    * **Immutability of On-Chain Hash**: The key here is that the NFT's commit hash itself is immutable on the blockchain. Even if the GitHub repo is tampered with, the blockchain provides an indisputable, immutable record of the *expected* and *last verified* state, acting as a cryptographic anchor.

- **Scenario: GitHub Service Outage:**
  - **Detection**: The KYA system's infrastructure will include global monitoring of GitHub's public status pages and direct API health checks.
  - **Handling**:
    * **Degraded Service**: During a GitHub outage, KYA verification processes that rely on fetching the off-chain data would be temporarily unavailable. Platforms would be advised to implement circuit breakers and display appropriate messages to users (e.g., "AI Agent verification temporarily unavailable, please try again later") rather than outright failures.
    * **Cached KYA Data (Limited Fallback):** For critical, frequently accessed permissions, platforms might implement a short-term, securely cached version of previously verified KYA data. However, this cache would have a very short Time-To-Live (TTL) and would only be used as a fallback for non-critical operations, as it wouldn't reflect real-time updates from GitHub.
    * **Redundant Off-Chain Storage (Future Enhancement):** For extreme resilience against single points of failure for the off-chain data, future iterations could explore distributing KYA data to decentralized storage networks (e.g., IPFS, Arweave) referenced in the NFT metadata. This would provide alternative, resilient retrieval points if GitHub is unavailable.

8.2 Solana Blockchain Related Edge Cases

While Solana is designed for high throughput and reliability, blockchain interactions can still encounter issues that the KYA system must gracefully handle.

- **Scenario: Solana Transaction Failure (Minting, Updating NFT Metadata):**
  * **Causes**: Common causes include insufficient SOL for gas fees (rent and compute units), temporary network congestion, invalid transaction parameters, compute budget exhaustion (if the instruction is too complex), or smart contract logic errors.
  * **Detection**: The KYA platform's backend (which initiates these transactions for KYA operations) will diligently monitor transaction status on Solana. If a transaction fails

or is dropped from the mempool, the Solana RPC node will return an error or the transaction will not appear in a confirmed block within a reasonable timeout.

    - **Handling**:
        - **Retry Mechanism**: Implement an intelligent retry mechanism with exponential backoff. This ensures that transient network issues or temporary congestion do not lead to permanent failures.
        - **Error Logging and Alerts**: Detailed logging of all transaction failures with specific Solana error codes. Alerts will be sent to the KYA platform's operations team for critical failures and, if user-initiated, clear notifications will be sent to the agent owner.
        - **User Feedback**: The KYA user interface will provide clear feedback to the user about failed operations and practical guidance on how to resolve them (e.g., "Insufficient SOL in your wallet, please top up" or "Network congested, please try again in a few minutes").
        - **Idempotency**: Smart contract functions (instructions) will be designed to be idempotent where possible, ensuring that retrying a transaction multiple times doesn't lead to unintended side effects (e.g., minting multiple NFTs for the same agent).

- **Scenario: Solana Network Congestion/Outage:**
    - **Detection**: Continuous monitoring of Solana's network health, including transaction confirmation times, block production, and overall RPC endpoint responsiveness.
    - **Handling**:
        - **Adaptive Retries/Backoff**: The KYA platform's transaction submission logic will automatically adapt by increasing retry attempts and backoff periods during periods of congestion to improve transaction success rates.
        - **Degraded Service Mode**: If congestion is severe or prolonged, the KYA platform might enter a "degraded service" mode. In this state, it may prioritize critical operations (e.g., identity verification) over less time-sensitive ones (e.g., new agent creation or metadata updates), ensuring core functionality remains operational.
        - **User Notification**: Proactive notification to users about network status and potential delays in on-chain operations.
        - **RPC Node Redundancy**: The KYA platform will connect to multiple Solana RPC nodes from diverse providers (e.g., Helius, QuickNode, Ankr) to minimize the impact of a single RPC endpoint failure or rate limiting.
- **Scenario: Smart Contract Bug or Vulnerability:**
    - **Detection**: Rigorous smart contract development lifecycle including: extensive unit and integration testing, formal verification methods, continuous security monitoring (on-chain activity analysis), and incentive-based bug bounty programs.
    - **Handling**:
        - **Upgradeability**: As discussed in Section 5.5, the KYA smart contracts on Solana are designed to be upgradable. In case of a critical bug or newly discovered vulnerability, a patched version of the program can be deployed by the securely

managed `upgrade_authority`.

- **Emergency Pause Function (with caution):** For extreme, system-threatening vulnerabilities, a pre-defined "pause" function could be included in the smart contract. This would allow the `upgrade_authority` to temporarily halt all or part of the contract's functionality, preventing further damage until a fix is deployed. This is a powerful tool to be used with extreme caution due to its centralization implications and should only be triggered for severe, unmitigable risks.
- **Incident Response Plan**: A clear and well-rehearsed incident response plan will be in place for rapid detection, containment, and remediation of smart contract vulnerabilities. This includes communication protocols to inform users and integrating platforms.

8.3 Data Consistency and Synchronization

Ensuring that the off-chain GitHub data and on-chain Solana NFT metadata remain synchronized and consistent is critical for the system's integrity.

* **Atomic Updates (Conceptual):** While true atomicity between disparate systems (GitHub and Solana) is challenging, the KYA system will strive for "eventual consistency" and utilize robust reconciliation mechanisms to address potential desynchronization.
* **Event-Driven Synchronization Pipeline:**
  - Changes to an agent's GitHub repository (new commits) will trigger webhooks configured within GitHub.
  - These webhooks will send notifications to a dedicated KYA backend service.
  - This backend service will then initiate a CI/CD-like pipeline that:
    1. Fetches the updated KYA data from GitHub at the new commit hash.
    2. Validates the updated KYA data against the latest schema (from the base KYA schema repo).
    3. If validation is successful, prepares and submits a Solana transaction to update the associated NFT's metadata with the new commit hash via the `update_kya_metadata` instruction. This ensures the on-chain pointer reflects the latest verified off-chain state.
* **Monitoring and Alerting for Discrepancies:** A dedicated monitoring service will continuously compare the `commit_hash` recorded on the Solana NFT with the latest verified `commit_hash` on the agent's GitHub repository. Any discrepancy or significant delay in synchronization will trigger automated alerts for manual intervention by the KYA operations team.
* **Periodic Reconciliation Scans:** Periodically (e.g., daily or weekly), the KYA system can perform full reconciliation scans. This involves iterating through all active KYA NFTs on Solana, fetching their recorded commit hashes, and then comparing these against the actual latest commit hashes of their respective GitHub repositories. Any inconsistencies found are flagged for resolution, ensuring long-term data integrity.

By proactively identifying and developing robust handling mechanisms for these error

conditions and edge cases, the KYA system aims to provide a resilient, reliable, and trustworthy identity infrastructure for AI agents, even when operating across centralized (GitHub) and decentralized (Solana) components. This commitment to resilience enhances the overall trustworthiness and adoption potential of the KYA framework.

9 Conclusion and Future Directions

The Know Your Agent (KYA) system presents a robust and innovative solution to the pressing need for standardized, secure, and verifiable identification for autonomous AI agents. By synergistically combining the immutable ledger capabilities of the Solana blockchain with the flexible, version-controlled data management of GitHub, KYA establishes a native identity layer that transcends the limitations of traditional IAM systems. This framework enables seamless, trustworthy, and auditable interactions for AI agents across diverse digital domains, from e-commerce to enterprise operations.

The detailed architecture outlined in this paper, encompassing a rich KYA schema, secure key management protocols tailored for both individual and enterprise users, and comprehensive strategies for schema evolution and error handling, demonstrates a system built for resilience and adaptability in a rapidly advancing technological landscape. KYA's unique approach addresses critical challenges of security, privacy, and accountability, fostering an environment where autonomous AI agents can flourish responsibly. We have shown how an agent can authenticate its ownership, prove its capabilities, and securely access sensitive information within predefined limits, all while maintaining an auditable trail.

By providing a universal, verifiable identity for AI agents, our KYA system is not just a technological innovation; it's a critical step towards building a responsible, transparent, and highly efficient agentic future for the digital economy.

9.1 Future Directions

The development of the KYA system is an ongoing endeavor, with several promising avenues for future research and implementation:

* **Open-Source Tooling and SDKs**: Developing comprehensive SDKs for various programming languages (Python, JavaScript, Rust) to facilitate easy integration by AI agent creators and platform developers. This includes libraries for KYA profile generation, signing operations, and verification modules.
* **Advanced Cryptographic Techniques**: Further exploring and integrating cutting-edge cryptographic techniques such as Zero-Knowledge Proofs (ZKPs) for enhanced privacy-preserving credential access and attribute verification. This would allow an agent to prove its eligibility (e.g., "I am authorized to spend up to $100") without revealing the underlying sensitive data (e.g., the exact spend limit, or the credit card number).
* **Decentralized Storage Redundancy**: Investigating and integrating with decentralized file storage solutions (e.g., IPFS, Arweave) to provide additional layers of resilience and censorship resistance for KYA data, offering alternative retrieval points beyond

reliance solely on GitHub. This would further decentralize the off-chain data layer.

* **Community Governance and DAO Model**: Establishing a decentralized autonomous organization (DAO) for the KYA system. This DAO could govern schema evolution, fund future development initiatives, manage critical smart contract upgrades, and resolve disputes, ensuring the KYA standard evolves in a community-driven manner.
* **Inter-Blockchain Compatibility**: Exploring bridges or cross-chain communication protocols to extend KYA's reach to other blockchain networks (e.g., Ethereum, Avalanche, Polygon), further enhancing interoperability and allowing KYA-identified agents to operate across a broader Web3 landscape.
* **Real-world Pilot Programs and Industry Partnerships**: Collaborating with leading AI companies, e-commerce platforms, and enterprise solutions providers to implement and test KYA in live environments. Gathering real-world feedback will be crucial for iterative refinement and demonstrating the system's practical utility and scalability.
* **Standardization Body Engagement**: Engaging with relevant industry standardization bodies (e.g., IEEE, W3C, Decentralized Identity Foundation) to promote KYA as a widely adopted standard for AI agent identity, fostering a unified and secure ecosystem.

By laying this foundational identity layer, the KYA system is not merely a technological advancement but a critical enabler for the secure, accountable, and widespread adoption of agentic AI, paving the way for its transformative impact on the global digital economy.

References

[1] Solana Documentation. Available at: https://docs.solana.com/

[2] GitHub Documentation. Available at: https://docs.github.com/

[3] Solana Program Library (SPL) Token Standard. Available at: https://spl.solana.com/token

[4] Metaplex Token Metadata Standard. Available at: https://docs.metaplex.com/token-metadata/overview

[5] The OAuth 2.0 Authorization Framework. Available at: https://oauth.net/2/

[6] OpenID Connect. Available at: https://openid.net/connect/

[7] Semantic Versioning 2.0.0. Available at: https://semver.org/

[8] Hardware Security Module (HSM) overview. Available at: https://en.wikipedia.org/wiki/Hardware_security_module

[9] Multi-Party Computation (MPC) Explained. Available at: https://coinmarketcap.com/academy/glossary/multi-party-computation-mpc

[10] Zero-Knowledge Proofs Explained. Available at: https://ethereum.org/en/zero-knowledge-proofs/

[11] Microsoft Active Directory Documentation. Available at: https://learn.microsoft.com/en-us/windows-server/identity/ad-ds/active-directory-domain-services

[12] Git Branching - Basic Branching and Merging. Available at: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging