Technical hypothesis for mining user data in the Solana system.

When solving a technical hypothesis for mining user data in the Solana system, we need to test several algorithms, which requires a long period of time to test such hypotheses; we can only write short smart contracts in the Rust language so that you can have an idea of what we are talking about and what we need build to bring our project to life.

ComposeDB is a graph database for Web3 applications based on Ceramic. ComposeDB allows developers to create and manage databases using a graphical interface, making the development process more intuitive and convenient. Additionally, ComposeDB integrates with other tools such as IPFS and Ethereum, allowing you to build more complex and scalable applications.

Data mining algorithm through Ceramic Nodes + CAIP-168 structures: IPLD timestamp proof + Ceramic HTTP API and combination with ComposeDB,
putting the results into IPFS to create data mining and a specific API including a smart contract to reward the user for processing
it is its data CID that can be implemented as follows:

1.Create a database in ComposeDB to store data that will be used in mining.

2.Create an application that will use the Ceramic HTTP API to retrieve data from Ceramic Nodes + CAIP-168: IPLD Timestamp Proof.
3.Processing the received data and recording the results into the ComposeDB database.
4.Creation of a smart contract in Rust, which will be used to reward users for processing their CID data.
5.Create an API that will be used to interact with the smart contract and provide rewards to users.
6.Placement of mining results in IPFS to ensure decentralized access to data.

7.Development of a mining mechanism that will be used to process the data and determine which users will be rewarded for processing their CID data. 8. Implementation of a mechanism for rewarding users for processing their CID data based on a smart contract.

9.Testing and optimization of the data mining algorithm to ensure maximum efficiency and performance.

To deploy this smart contract, which must include the RUST programming language, we use the Solana platform.
Solana is a high-performance blockchain platform that uses Proof of History (PoH) consensus and can process up to 65,000 transactions per second. Solana also supports the Rust programming language, making it an ideal choice for developing smart contracts in Rust.

Although Ethereum also supports the Rust programming language, it has performance and scalability limitations that can lead to problems processing large numbers of transactions. Cosmos and Poligon may also be good choices, but they do not support the Rust programming language.

Binance Smart Chain also supports Rust, but it has scalability and security limitations.

Thus, I would recommend using Solana to deploy this smart contract, which should include the Rust programming language.
Solana provides high performance and scalability, and also supports Rust, making it an ideal choice for developing smart contracts in Rust.
In addition, users of Helium, a project that has moved to Solana, will be able to freely mine their data, thereby we will be able to attract from 600,000 to 1 million users to the project in the future of the project's deployment.

To implement a smart contract on Solana using the Rust programming language and the Web3.js library to interact with the contract,
you can use the following approach:

1.Creation of a smart contract in the Rust programming language, which will be used for authorization and authentication of users.
2. Implementation of the client part of the system using the Web3.js library, which will be used to interact with a smart contract on Solana.
3.Creating a database in ComposeDB to store data that will be used in mining.

4.Implementation of a data mining mechanism that will be used to process data and determine

which users will be rewarded for processing their CID data.
5.Placement of mining results in IPFS to ensure decentralized access to data.

6.Creation of a decentralized system where each user will have access to their data to be stored in IPFS.
7. Implementation of a mechanism for rewarding users for processing their CID data based on a smart contract.
8.Testing and optimization of the data mining algorithm to ensure maximum efficiency and performance.

Thus, using the Rust programming language and the Web3.js library, it is possible to implement a decentralized system of authorization and authentication of users on the Solana blockchain, as well as create a decentralized data storage system with the ability to mine data and reward users for processing it.

We combine the creation of a smart contract in Solana using the Rust programming language and the Web3.js library with the ComposeDB graphical database,
which is intended for use in Ceramic-based Web3 applications. ComposeDB allows developers to create and manage databases using a graphical interface, making the development process more intuitive and convenient. In addition, ComposeDB integrates with other tools such as IPFS, allowing you to build decentralized applications on top of Ceramic.

Thus, you can use ComposeDB to create a database that will be used in a Solana smart contract to store data related to user authorization and authentication. This will simplify the database development and management process, as well as ensure its integration with other tools such as IPFS.

Additionally, a data mining mechanism can be used to process the data and determine which users will be rewarded for processing their CID data. Mining results can be stored in ComposeDB, which will provide quick and convenient access to data for users.

Thus, combining the creation of a smart contract in Solana using the Rust programming language and the Web3.js library with the ComposeDB graphical database will allow you to create a decentralized system for authorization and authentication of users based on Ceramic with the ability to mine data and reward users for processing it.

The final code of a smart contract on Solana, which implements a system for authorization and authentication of users with a mechanism for mining data and rewarding users for processing it, may look like this:

rust

Copy

```
// Import the necessary libraries use
solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint,
    entrypoint::ProgramResult,
    msg,
    program_error::ProgramError,
    pubkey::Pubkey,
};

// Import the library for working with IPFS use
ipfs_api::IpfsApi;

// Import the library to work with ComposeDB
use compose_db::{ComposeDB, ComposeDBError, ComposeDBResult};

// Define the structure of the smart contract
pub struct AuthContract {}

// Define methods for the smart contract structure impl
AuthContract {
    // Method for checking pub user authorization fn
    authorize_user(user_pubkey: &Pubkey) -> bool {
        // There should be logic here to check the user's authorization // Return
        true if the user is authorized, and false if not true

    }

    // Method for user authentication
    pub fn authenticate_user(user_pubkey: &Pubkey, user_signature: &[u8]) -> bool {
        // There should be user authentication logic here
```

```
        // Return true if the user is authenticated, false if not true

    }

    // Method for data mining
    pub fn mine_data(cid: &str) -> ComposeDBResult<()> {
        // Create a ComposeDB instance let mut
        db = ComposeDB::new("auth_db");

        // Get data from IPFS
        let ipfs = IpfsApi::new("localhost", 5001);
        let data = ipfs.cat(cid)?;

        // Perform data mining
        // There should be data mining logic here

        // Save the mining results in ComposeDB
        db.insert(cid, "mined_data", "result")?;

        Ok(())
    }

    // Method for rewarding users
    pub fn reward_user(user_pubkey: &Pubkey, amount: u64) -> ProgramResult {
        // There should be logic here for rewarding users // Return Ok if the
        user was rewarded successfully Ok(())

    }
}

// Define the entry point for the smart contract
entrypoint!(process_instruction);

// Define a method for processing instructions fn
process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) - > ProgramResult {
    // Get information about the current account let
    accounts_iter = &mut accounts.iter();
    let account = next_account_info(accounts_iter)?;

    // Check that the current account is authorized if !
    AuthContract::authorize_user(account.key) {
        return Err(ProgramError::InvalidAccountData);
    }
```

```
    // Get information about the user and his signature let
    user_account = next_account_info(accounts_iter)?; let
    user_signature = next_account_info(accounts_iter)?;

    // Check that the user is authenticated
    if !AuthContract::authenticate_user(user_account.key, user_signature.data) {
        return Err(ProgramError::InvalidAccountData);
    }

    // Get the CID of the data from the instruction
    let cid = String::from_utf8(instruction_data.to_vec()).unwrap();

    // Mining data
    AuthContract::mine_data(&cid)?;

    // Reward the user
    AuthContract::reward_user(user_account.key, 100)?;

    Ok(())
}
```

This smart contract defines the AuthContract structure, which contains methods for checking authorization and authenticating users, as well as for mining data and rewarding users for processing it. At the process_instruction entry point, we check that the current account is authorized, and then get information about the user and his signature. After this, we verify that the user is authenticated and perform data mining and reward the user.

For data mining, we use the IPFS library to obtain data by CID and execute the data mining logic. We then store the mining results in ComposeDB using the ComposeDB library. To reward users, we execute the reward logic and return Ok if the user was rewarded successfully.

This smart contract allows you to create a decentralized system for authorization and authentication of users based on Ceramic with the ability to mine data and reward users for processing it.

The final code of the smart contract on Solana, which implements a system of authorization and authentication of users with a mechanism for mining data and rewarding users for their processing, and how data mining will occur may look like this:

rust

Copy

```
// Import the necessary libraries use
solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint::ProgramResult,
```

```
    msg,
    program_error::ProgramError,
    pubkey::Pubkey,
};

// Import the library for working with IPFS use
ipfs_api::IpfsApi;

// Import the library to work with ComposeDB
use compose_db::{ComposeDB, ComposeDBError, ComposeDBResult};

// Define the structure of the smart contract
pub struct AuthContract {}

// Define methods for the smart contract structure impl
AuthContract {
    // Method for checking pub user authorization fn
    authorize_user(user_pubkey: &Pubkey) -> bool {
        // There should be logic here to check the user's authorization // Return
        true if the user is authorized, and false if not true

    }

    // Method for user authentication
    pub fn authenticate_user(user_pubkey: &Pubkey, user_signature: &[u8]) -> bool {
        // There should be user authentication logic here
        // Return true if the user is authenticated, false if not true

    }

    // Method for data mining
    pub fn mine_data(cid: &str) -> ComposeDBResult<()> {
        // Create a ComposeDB instance let mut
        db = ComposeDB::new("auth_db");

        // Get data from IPFS
        let ipfs = IpfsApi::new("localhost", 5001);
        let data = ipfs.cat(cid)?;

        // Perform data mining
        // There should be data mining logic here

        // Save the mining results in ComposeDB
        db.insert(cid, "mined_data", "result")?;

        Ok(())
    }
```

```
    // Method for rewarding users
    pub fn reward_user(user_pubkey: &Pubkey, amount: u64) -> ProgramResult {
        // There should be logic here for rewarding users // Return Ok if the
        user was rewarded successfully Ok(())

    }
}

// Define the entry point for the smart contract
entrypoint!(process_instruction);

// Define a method for processing instructions fn
process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) - > ProgramResult {
    // Get information about the current account let
    accounts_iter = &mut accounts.iter();
    let account = next_account_info(accounts_iter)?;

    // Check that the current account is authorized if !
    AuthContract::authorize_user(account.key) {
        return Err(ProgramError::InvalidAccountData);
    }

    // Get information about the user and his signature let
    user_account = next_account_info(accounts_iter)?; let
    user_signature = next_account_info(accounts_iter)?;

    // Check that the user is authenticated
    if !AuthContract::authenticate_user(user_account.key, user_signature.data) {
        return Err(ProgramError::InvalidAccountData);
    }

    // Get the CID of the data from the instruction
    let cid = String::from_utf8(instruction_data.to_vec()).unwrap();

    // Mining data
    AuthContract::mine_data(&cid)?;

    // Reward the user
    AuthContract::reward_user(user_account.key, 100)?;

    Ok(())
}
```

In this smart contract, data mining is performed in the mine_data method.

After getting the CID data from IPFS, we call this method passing it the CID as an argument.

Inside the mine_data method, we create a ComposeDB instance and retrieve data from IPFS using the IPFS library. We then execute the data mining logic. The logic of data mining may vary depending on the specific problem we are solving. For example, we can use machine learning to process data and produce results.

After performing data mining, we store the results in ComposeDB using the insert method.

We pass the CID, the key "mined_data" and the value "result" as arguments to the insert method.

Data mining is an important part of decentralized applications as it allows data to be processed without the need for centralized management. IN
In this smart contract, we use the IPFS and ComposeDB libraries to process and store data, which allows us to create decentralized applications with high performance and scalability.

```
// Method for data mining
pub fn mine_data(cid: &str) -> ComposeDBResult<()> {
    // Create a ComposeDB instance let mut
    db = ComposeDB::new("auth_db");

    // Get data from IPFS
    let ipfs = IpfsApi::new("localhost", 5001);
    let data = ipfs.cat(cid)?;

    // Perform data mining
    // There should be data mining logic here

    // Save the mining results in ComposeDB
    db.insert(cid, "mined_data", "result")?;

    Ok(())
}
```

In this mine_data method, we use the IPFS library to obtain data by CID and the ComposeDB library to save the results of data mining. However, there is no data mining logic here.

The logic of data mining may vary depending on the specific problem we are solving.

For example, if we want to process data using machine learning, then we can use libraries such as TensorFlow or PyTorch to create a machine learning model and process the data.

After processing the data, we can store the results in ComposeDB using the insert method. We can pass the CID, the key "mined_data" and the value "result" as arguments to the insert method.

An example of data mining logic for image processing using the TensorFlow library might look like this:
rust

Copy

```
// Method for data mining
pub fn mine_data(cid: &str) -> ComposeDBResult<()> {
    // Create a ComposeDB instance let mut
    db = ComposeDB::new("auth_db");

    // Get data from IPFS
    let ipfs = IpfsApi::new("localhost", 5001);
    let data = ipfs.cat(cid)?;

    // Load the machine learning model let model =
    tf::Graph::new().import(&model_path)?;

    // Process data using a machine learning model let result = model

        . session(&sess_options)?
        . run(&inputs, &outputs, &targets)?;

    // Save the mining results in ComposeDB
    db.insert(cid, "mined_data", result)?;

    Ok(())
}
```

In this example, we load a machine learning model from a file and use it to process the data.

We pass data to the model as input parameters and receive results as output parameters.

We then store the mining results in ComposeDB using the insert method.

What data do we receive and try to process // We receive data from IPFS

```
    let ipfs = IpfsApi::new("localhost", 5001);
    let data = ipfs.cat(cid)?;

    // Load the machine learning model let model =
    tf::Graph::new().import(&model_path)?;

    // Process data using a machine learning model let result = model

        . session(&sess_options)?
        . run(&inputs, &outputs, &targets)?;
```

In this example, we retrieve data from IPFS by CID and process it using a machine learning model. The specific data we receive and process depends on the specific task we are solving.

For example, if we are solving an image classification problem, then we can retrieve images from IPFS and process them using a machine learning model to determine which class they belong to. In this case, the data we receive and process will be images.

If we are solving a text data processing problem, then we can get text data from IPFS and process it using a machine learning model.
to perform tasks such as text classification or text generation. In this case, the data we receive and process will be text.
In general, we can take any data from IPFS and process it using a machine learning model to perform various tasks such as classification, clustering, regression, etc.

Data mining, for which the user is rewarded by receiving Solana tokens, occurs in such a way that machine learning identifies the user data that is in the block and having found this data only then the user is rewarded with a token, but if it happens that the user data is not found, then the tokens are not charged for a given user

```
// Get data from IPFS
  let ipfs = IpfsApi::new("localhost", 5001);
  let data = ipfs.cat(cid)?;

  // Load the machine learning model let model =
  tf::Graph::new().import(&model_path)?;

  // Process data using a machine learning model let result = model

    . session(&sess_options)?
    . run(&inputs, &outputs, &targets)?;
```

In this example, we use a machine learning model to process data obtained from IPFS.

However, in your description of the problem, data mining occurs not only on the basis of data processing, but also on the basis of its presence in the block.
To solve this problem, we can use the following algorithm: Receive user data
from the Solana block.
We are looking for the CID corresponding to the user data in IPFS.
If the CID is found, then we process the data using a machine learning model as shown in the previous example.
If the CID is not found, then we do not award tokens to the user.

An example code to implement this algorithm might look like this:

rust

Copy

```
// Method for data mining
pub fn mine_data(user_id: &str, block_data: &BlockData) -> ComposeDBResult<()> {
    // Create a ComposeDB instance let mut
    db = ComposeDB::new("auth_db");

    // Get the CID from the Solana block let
    cid = block_data.get_cid(user_id)?;

    // If the CID is found, then process the data if
    let Some(cid) = cid {
        // Get data from IPFS
        let ipfs = IpfsApi::new("localhost", 5001);
        let data = ipfs.cat(cid)?;

        // Load the machine learning model let model =
        tf::Graph::new().import(&model_path)?;

        // Process data using a machine learning model let result = model

            . session(&sess_options)?
            . run(&inputs, &outputs, &targets)?;

        // Credit tokens to the user let tokens =
        calculate_tokens(result);
        block_data.add_tokens(user_id, tokens)?;

        // Save the mining results in ComposeDB
        db.insert(cid, "mined_data", result)?;
    }

    Ok(())
}
```

In this example, we get the CID from a Solana block using the get_cid method, which returns the CID corresponding to the user data if found, or None if the user data is not found. If the CID is found, then we process the data using a machine learning model and assign tokens to the user using the add_tokens method. If the CID is not found, then we do not award tokens to the user.

Describe in more detail the structure of identifying CIDs using this algorithm. In this example, we get the CID from the Solana block using the get_cid method, which returns the CID corresponding to the user data if it is found, or None if the user data is not found. If the CID is found, then we process the data using a machine learning model and assign tokens to the user using the add_tokens method. If the CID is not found, then we do not award tokens to the user.

The specific CID detection structure depends on how the user data is stored in the Solana block.

However, in general, we can use the following algorithm to identify the CID: Receive a Solana block containing the user data.
Retrieving user data from the block. We
calculate the hash of the user data.
Using the user data hash, we look for the CID in IPFS.
If the CID is found, then we return it. If the CID is not found, then we return None.

An example code to implement this algorithm might look like this:

rust

Copy

```
// Method to get the CID corresponding to the user data pub fn
get_cid(user_id: &str, block: &SolanaBlock) -> Option<String> {
    // Retrieving user data from the block let
    user_data = block.get_user_data(user_id)?;

    // Calculate the user data hash let hash =
    calculate_hash(user_data);

    // Looking for CID in IPFS
    let ipfs = IpfsApi::new("localhost", 5001);
    let cid = ipfs.resolve(&hash)?;

    // If the CID is found, then return it. If the CID is not found, then we return None. cid

}
```

In this example, we use the get_user_data method to retrieve user data from a Solana block. We then calculate a hash of the user's data using the calculate_hash function. Next, we look up the CID in IPFS using the resolve method. If the CID is found, then we return it. If the CID is not found, then we return None.

The specific smart contract for CID mining will depend on how the user data is stored in the Solana block and how it is linked to the CID in IPFS.
However, in general, we can use the following smart contract to mine CID:

rust

Copy

```
// Smart contract for mining CID pub
struct MiningContract {
    // IPFS node address
    ipfs_address: String,
}
```

```
impl MiningContract {
    // Method for mining CID
    pub fn mine_cid(&self, user_id: &str, block: &SolanaBlock) -> ProgramResult {
        // Get the CID corresponding to the user data let cid =
        self.get_cid(user_id, block)?;

        // If the CID is found, then we award tokens to the user if
        let Some(cid) = cid {
            self.add_tokens(user_id, cid)?;
        }

        Ok(())
    }

    // Method to get the CID corresponding to the user data
    fn get_cid(&self, user_id: &str, block: &SolanaBlock) -> ProgramResult<Option<String>> {
        // Retrieving user data from the block let
        user_data = block.get_user_data(user_id)?;

        // Calculate the user data hash let hash =
        calculate_hash(user_data);

        // Looking for CID in IPFS
        let ipfs = IpfsApi::new(&self.ipfs_address, 5001); let
        cid = ipfs.resolve(&hash)?;

        // If the CID is found, then return it. If the CID is not found, then we return None.
        Ok(cid)
    }

    // Method for assigning tokens to the user
    fn add_tokens(&self, user_id: &str, cid: String) -> ProgramResult {
        // Get the number of tokens that need to be credited to the user let tokens =
        calculate_tokens(cid);

        // Assign tokens to the user
        let mut user_account = self.get_user_account(user_id)?;
        user_account.tokens += tokens;
        self.save_user_account(user_id, &user_account)?;

        Ok(())
    }

    // Method for getting user balance
    fn get_user_account(&self, user_id: &str) -> ProgramResult<UserAccount> {
        // Get user data from the blockchain let
        account_data = get_account_data(user_id)?;
```

```
        // Deserialize user data
        let user_account = UserAccount::deserialize(&account_data)?;

        Ok(user_account)
    }

    // Method for saving the user's balance
    fn save_user_account(&self, user_id: &str, user_account: &UserAccount) -> ProgramResult {
        // Serialize user data let account_data =
        user_account.serialize()?;

        // Save user data in the blockchain
        save_account_data(user_id, &account_data)?;

        Ok(())
    }
}
```

In this example, we use the MiningContract framework to implement a smart contract for CID mining. The mine_cid method is used to mine CID and assign tokens to the user. The get_cid method is used to obtain the CID corresponding to the user data. The add_tokens method is used to assign tokens to the user. The get_user_account and save_user_account methods are used to retrieve and save the user's balance on the blockchain.

```
 // Import the necessary libraries use
solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint::ProgramResult,
    msg,
    program_error::ProgramError,
    pubkey::Pubkey,
};

// Import the library for working with IPFS use
ipfs_api::IpfsApi;

// Import the library to work with ComposeDB
use compose_db::{ComposeDB, ComposeDBError, ComposeDBResult};

// Define the structure of the smart contract
pub struct AuthContract {}

// Define methods for the smart contract structure impl
AuthContract {
    // Method for checking pub user authorization fn
    authorize_user(user_pubkey: &Pubkey) -> bool {
```

```
        // There should be logic here to check the user's authorization // Return
        true if the user is authorized, and false if not true

    }

    // Method for user authentication
    pub fn authenticate_user(user_pubkey: &Pubkey, user_signature: &[u8]) -> bool {
        // There should be user authentication logic here
        // Return true if the user is authenticated, false if not true

    }

    // Method for data mining
    pub fn mine_data(cid: &str) -> ComposeDBResult<()> {
        // Create a ComposeDB instance let mut
        db = ComposeDB::new("auth_db");

        // Get data from IPFS
        let ipfs = IpfsApi::new("localhost", 5001);
        let data = ipfs.cat(cid)?;

        // Perform data mining
        // There should be data mining logic here

        // Save the mining results in ComposeDB
        db.insert(cid, "mined_data", "result")?;

        Ok(())
    }

    // Method for rewarding users
    pub fn reward_user(user_pubkey: &Pubkey, amount: u64) -> ProgramResult {
        // There should be logic here for rewarding users // Return Ok if the
        user was rewarded successfully Ok(())

    }
}

// Define the entry point for the smart contract
entrypoint!(process_instruction);

// Define a method for processing instructions fn
process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) - > ProgramResult {
    // Get information about the current account
```

```
    let accounts_iter = &mut accounts.iter();
    let account = next_account_info(accounts_iter)?;

    // Check that the current account is authorized if !
    AuthContract::authorize_user(account.key) {
        return Err(ProgramError::InvalidAccountData);
    }

    // Get information about the user and his signature let
    user_account = next_account_info(accounts_iter)?; let
    user_signature = next_account_info(accounts_iter)?;

    // Check that the user is authenticated
    if !AuthContract::authenticate_user(user_account.key, user_signature.data) {
        return Err(ProgramError::InvalidAccountData);
    }

    // Get the CID of the data from the instruction
    let cid = String::from_utf8(instruction_data.to_vec()).unwrap();

    // Mining data
    AuthContract::mine_data(&cid)?;

    // Reward the user
    AuthContract::reward_user(user_account.key, 100)?;

    Ok(())
}
```

The final smart contract code with the CID mining code inserted looks like this:

rust

Copy

```
// Import the necessary libraries use
solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint::ProgramResult,
    msg,
    program_error::ProgramError,
    pubkey::Pubkey,
};

// Import the library for working with IPFS use
ipfs_api::IpfsApi;
```

```
// Import the library to work with ComposeDB
use compose_db::{ComposeDB, ComposeDBError, ComposeDBResult};

// Define the structure of the smart contract
pub struct AuthContract {}

// Define methods for the smart contract structure impl
AuthContract {
    // Method for checking pub user authorization fn
    authorize_user(user_pubkey: &Pubkey) -> bool {
        // There should be logic here to check the user's authorization // Return
        true if the user is authorized, and false if not true

    }

    // Method for user authentication
    pub fn authenticate_user(user_pubkey: &Pubkey, user_signature: &[u8]) -> bool {
        // There should be user authentication logic here
        // Return true if the user is authenticated, false if not true

    }

    // Method for data mining
    pub fn mine_data(cid: &str) -> ComposeDBResult<()> {
        // Create a ComposeDB instance let mut
        db = ComposeDB::new("auth_db");

        // Get data from IPFS
        let ipfs = IpfsApi::new("localhost", 5001);
        let data = ipfs.cat(cid)?;

        // Perform data mining
        // There should be data mining logic here

        // Save the mining results in ComposeDB
        db.insert(cid, "mined_data", "result")?;

        Ok(())
    }

    // Method for rewarding users
    pub fn reward_user(user_pubkey: &Pubkey, amount: u64) -> ProgramResult {
        // There should be logic here for rewarding users // Return Ok if the
        user was rewarded successfully Ok(())

    }

    // Method for mining CID
```

```
    pub fn mine_cid(ipfs_address: &str, user_id: &str, block: &SolanaBlock) -> ProgramResult {
        // Create an IPFS API instance
        let ipfs = IpfsApi::new(ipfs_address, 5001);

        // Get user data from the block let user_data =
        block.get_user_data(user_id)?;

        // Calculate the user data hash let hash =
        calculate_hash(user_data);

        // Looking for CID in IPFS
        let cid = ipfs.resolve(&hash)?;

        // If the CID is found, then we award tokens to the user if
        let Some(cid) = cid {
            AuthContract::reward_user(user_id, 100)?;
        }

        Ok(())
    }
}

// Define the entry point for the smart contract
entrypoint!(process_instruction);

// Define a method for processing instructions fn
process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) - > ProgramResult {
    // Get information about the current account let
    accounts_iter = &mut accounts.iter();
    let account = next_account_info(accounts_iter)?;

    // Check that the current account is authorized if !
    AuthContract::authorize_user(account.key) {
        return Err(ProgramError::InvalidAccountData);
    }

    // Get information about the user and his signature let
    user_account = next_account_info(accounts_iter)?; let
    user_signature = next_account_info(accounts_iter)?;

    // Check that the user is authenticated
    if !AuthContract::authenticate_user(user_account.key, user_signature.data) {
        return Err(ProgramError::InvalidAccountData);
    }
```

```
    // Get the CID of the data from the instruction
    let cid = String::from_utf8(instruction_data.to_vec()).unwrap();

    // Perform CID mining
    AuthContract::mine_cid("localhost", user_account.key.to_string(), block)?;

    Ok(())
}
```

In this example, we added the mine_cid method for mining CID and assigning tokens to the user.
The mine_cid method uses the IPFS API to obtain the CID corresponding to the user data and calls the reward_user method to reward tokens to the user.

how to insert this smart contract into the API to check at login and mine data during authorization and authentication of users

To insert this smart contract into the API, you must complete the following steps:

1.Create a Solana program using the Rust language and Solana SDK. The program must define the methods described in the smart contract, as well as the entry point for processing instructions.

2. Place the compiled binary file of the program on the Solana blockchain.
3.Create an API that will interact with the Solana blockchain via the Solana SDK. The API must provide methods for authorizing and authenticating users, as well as for data mining.

4.In the user authorization and authentication methods, call the appropriate smart contract methods to check authorization and authentication.
5.In the data mining method, call the smart contract data mining method to perform mining and save the results in the blockchain.
6.When assigning tokens to the user, call the token accrual method of the smart contract to save the transaction in the blockchain.
7. Ensure the security of the API and Solana program by using appropriate security measures such as validating input data, restricting access to methods, etc.

Thus, inserting a given smart contract into an API requires creating a Solana program, placing it on the blockchain, creating an API, and interacting with the blockchain through the Solana SDK.

The final smart contract code for authorization, authentication and data mining looks like this:

rust

Copy

```
// Import the necessary libraries use
solana_program::{
```

```
    account_info::{next_account_info, AccountInfo},
    entrypoint::ProgramResult,
    msg,
    program_error::ProgramError,
    pubkey::Pubkey,
};

// Import the library for working with IPFS use
ipfs_api::IpfsApi;

// Import the library to work with ComposeDB
use compose_db::{ComposeDB, ComposeDBError, ComposeDBResult};

// Define the structure of the smart contract
pub struct AuthContract {}

// Define methods for the smart contract structure impl
AuthContract {
    // Method for checking pub user authorization fn
    authorize_user(user_pubkey: &Pubkey) -> bool {
        // There should be logic here to check the user's authorization // Return
        true if the user is authorized, and false if not true

    }

    // Method for user authentication
    pub fn authenticate_user(user_pubkey: &Pubkey, user_signature: &[u8]) -> bool {
        // There should be user authentication logic here
        // Return true if the user is authenticated, false if not true

    }

    // Method for data mining
    pub fn mine_data(cid: &str) -> ComposeDBResult<()> {
        // Create a ComposeDB instance let mut
        db = ComposeDB::new("auth_db");

        // Get data from IPFS
        let ipfs = IpfsApi::new("localhost", 5001);
        let data = ipfs.cat(cid)?;

        // Perform data mining
        // There should be data mining logic here

        // Save the mining results in ComposeDB
        db.insert(cid, "mined_data", "result")?;

        Ok(())
```

```
    }

    // Method for rewarding users
    pub fn reward_user(user_pubkey: &Pubkey, amount: u64) -> ProgramResult {
        // There should be logic here for rewarding users // Return Ok if the
        user was rewarded successfully Ok(())

    }

    // Method for mining CID
    pub fn mine_cid(ipfs_address: &str, user_id: &str, block: &SolanaBlock) -> ProgramResult {
        // Create an IPFS API instance
        let ipfs = IpfsApi::new(ipfs_address, 5001);

        // Get user data from the block let user_data =
        block.get_user_data(user_id)?;

        // Calculate the user data hash let hash =
        calculate_hash(user_data);

        // Looking for CID in IPFS
        let cid = ipfs.resolve(&hash)?;

        // If the CID is found, then we award tokens to the user if
        let Some(cid) = cid {
            AuthContract::reward_user(user_id, 100)?;
        }

        Ok(())
    }
}

// Define the entry point for the smart contract
entrypoint!(process_instruction);

// Define a method for processing instructions fn
process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) - > ProgramResult {
    // Get information about the current account let
    accounts_iter = &mut accounts.iter();
    let account = next_account_info(accounts_iter)?;

    // Check that the current account is authorized if !
    AuthContract::authorize_user(account.key) {
        return Err(ProgramError::InvalidAccountData);
```

```
    }

    // Get information about the user and his signature let
    user_account = next_account_info(accounts_iter)?; let
    user_signature = next_account_info(accounts_iter)?;

    // Check that the user is authenticated
    if !AuthContract::authenticate_user(user_account.key, user_signature.data) {
        return Err(ProgramError::InvalidAccountData);
    }

    // Get the CID of the data from the instruction
    let cid = String::from_utf8(instruction_data.to_vec()).unwrap();

    // Perform CID mining
    AuthContract::mine_cid("localhost", user_account.key.to_string(), block)?;

    Ok(())
}
```

To interact with a smart contract via the API, you need to create the appropriate methods,

which will call smart contract methods.
For example, a method to authorize a user might look like this:

rust

Copy

```
fn authorize_user(user_pubkey: &Pubkey) -> bool {
    // Get information about the current account let
    account = get_account_info(user_pubkey);

    // Call the smart contract authorization method
    AuthContract::authorize_user(account.key)
}
```

Similarly, the method for user authentication might look like this:

rust

Copy

```
fn authenticate_user(user_pubkey: &Pubkey, user_signature: &[u8]) -> bool {
    // Get information about the current account and its signature let
    account = get_account_info(user_pubkey);
    let signature = get_signature_info(user_signature);

    // Call the smart contract authentication method
    AuthContract::authenticate_user(account.key, signature.data)
```

}
A method for mining data might look like this:

rust

Copy

```
fn mine_data(cid: &str) -> Result<(), String> {
    // Call the smart contract data mining method match
    AuthContract::mine_data(cid) {
        Ok(_) => Ok(()),
        Err(e) => Err(format!("Error mining data: {:?}", e)),
    }
}
```

Thus, to interact with a smart contract via the API, it is necessary to create the appropriate methods,
which will call smart contract methods and ensure secure interaction.

Let's combine this smart contract with Ceramic in IPFS for a decentralized ecosystem

To combine this smart contract with Ceramic in IPFS to create a decentralized ecosystem,

you can use the following approach:

1.Creation of a smart contract on the blockchain that will manage access to data in Ceramic and IPFS. In this smart contract, methods can be defined for authorization, authentication, data mining, and rewarding users as described above.

2.Creating an application that will interact with a smart contract via an API. This application can be written in any programming language and use any framework to create the API. In this application, you can define methods to interact with the smart contract as described above.

3.Creating a client to work with Ceramic and IPFS. To work with Ceramic you can use the @ceramicnetwork/http-client library, and to work with IPFS you can use the ipfs-http-client library. You can define methods in the client to read and write data to Ceramic and IPFS.

4. Integration of the client with the application. To do this, you can use the API created in the application and call the appropriate client methods to read and write data to Ceramic and IPFS.

5.Creating a data schema in Ceramic. To do this, you can use the @ceramicnetwork/cli library and define a data schema in JSON format.
The data schema must match the structure of the data that will be stored in Ceramic.

6.Create a record in Ceramic. To do this, you can use client methods to write data to Ceramic.

The data must conform to the data schema defined in Ceramic.

7.Create an entry in IPFS. To do this, you can use client methods to write data to IPFS.

Data must be encrypted and signed to ensure the security of data storage and transmission.

8.Updating a record in Ceramic. To do this, you can use client methods to update data in Ceramic.

The data must conform to the data schema defined in Ceramic.

9.Updating an entry in IPFS. To do this, you can use client methods to update data in IPFS.

Data must be encrypted and signed to ensure the security of data storage and transmission.

10.Reading a record from Ceramic. To do this, you can use client methods to read data from Ceramic.

The data must conform to the data schema defined in Ceramic.

11.Reading a record from IPFS. To do this, you can use client methods to read data from IPFS.

The data must be decrypted and authenticated to ensure the data is safe to read.

This creates a decentralized ecosystem in which a smart contract on the blockchain controls access to data in Ceramic and IPFS, and the application and client provide interaction with this data. This ecosystem can be used to create various decentralized applications such as social networks, data markets, etc.