MONDAY, 9 SEP, 2024 AT 1:45 AM

. Aiken-lang smart contract development Research

MILESTONE 2:

1. Aiken-lang features :

Aiken is a smart contract development language specifically designed for the Cardano blockchain. It aims to simplify the process of writing, testing, and deploying smart contracts on Cardano, addressing some of the complexities associated with Plutus, the original smart contract platform for Cardano. Here are the main features of Aiken:

1. Simplicity and Readability:

- Aiken emphasizes simplicity and readability, making it easier for developers to understand and write smart contracts compared to other languages like Plutus or Solidity.
  - It aims to reduce the cognitive load on developers by providing clear syntax and semantics.

2. Off-chain and On-chain Code Separation:

- Aiken cleanly separates off-chain code (the code that interacts with the blockchain) from on-chain code (the code executed by the blockchain).
  - This separation helps in managing the complexity of smart contracts and ensures that developers can focus on specific aspects of their contracts.

3. Performance Optimizations:

- Aiken is designed to optimize smart contracts for performance, minimizing execution costs on the blockchain.
  - It leverages efficient compilation and low-level optimizations to ensure that contracts run smoothly and cost-effectively.

4. Built-in Testing and Debugging Tools:

- Aiken provides integrated testing frameworks, allowing developers to test their contracts thoroughly before deployment.
  - Debugging tools are also included to help developers identify and fix issues in their contracts.

5. Type Safety and Functional Programming:

- Aiken is a functional programming language, emphasizing immutability and type safety, which reduces common programming errors.
  - This approach enhances security by preventing unintended behaviors and vulnerabilities in smart contracts.

6. Interoperability with Cardano Ecosystem:

- Aiken is designed to be fully compatible with the Cardano ecosystem, supporting the Cardano-node

and Cardano wallet integration.
    - This makes it easier to deploy and interact with smart contracts on the Cardano blockchain.

7. Low Learning Curve for Developers:

- Aiken is intended to have a lower learning curve for developers, especially those familiar with functional programming languages.
    - Its design philosophy is to be intuitive and accessible, which helps in broadening the developer base for Cardano.

8. Efficient Resource Usage:

- Contracts written in Aiken are designed to be resource-efficient, optimizing the use of Cardano's UTXO model and minimizing transaction fees.
    - This is crucial for maintaining the scalability and sustainability of the Cardano network.

9. Formal Verification Support:

- Aiken supports formal verification, allowing developers to prove the correctness of their smart contracts mathematically.
    - This feature enhances the reliability and security of contracts, especially in critical applications like finance.

10. Open Source and Community-Driven:

- Aiken is open-source, encouraging contributions and improvements from the Cardano community.
    - The language evolves through community feedback and collaboration, ensuring that it meets the needs of developers.

These features make Aiken an attractive choice for developers looking to create smart contracts on Cardano, offering a balance of usability, performance, and security.

2. List of all functions of aiken

Aiken is a smart contract development language specifically designed for the Cardano blockchain. It aims to simplify the process of writing, testing, and deploying smart contracts on Cardano, addressing some of the complexities associated with Plutus, the original smart contract platform for Cardano. Here are the main features of Aiken:

1. Simplicity and Readability:

- Aiken emphasizes simplicity and readability, making it easier for developers to understand and write smart contracts compared to other languages like Plutus or Solidity.
- It aims to reduce the cognitive load on developers by providing clear syntax and semantics.

2. Off-chain and On-chain Code Separation:

- Aiken cleanly separates off-chain code (the code that interacts with the blockchain) from on-chain code (the code executed by the blockchain).

- This separation helps in managing the complexity of smart contracts and ensures that developers can focus on specific aspects of their contracts.

3. Performance Optimizations:

- Aiken is designed to optimize smart contracts for performance, minimizing execution costs on the blockchain.
- It leverages efficient compilation and low-level optimizations to ensure that contracts run smoothly and cost-effectively.

4. Built-in Testing and Debugging Tools:

- Aiken provides integrated testing frameworks, allowing developers to test their contracts thoroughly before deployment.
- Debugging tools are also included to help developers identify and fix issues in their contracts.

5. Type Safety and Functional Programming:

- Aiken is a functional programming language, emphasizing immutability and type safety, which reduces common programming errors.
- This approach enhances security by preventing unintended behaviors and vulnerabilities in smart contracts.

6. Interoperability with Cardano Ecosystem:

- Aiken is designed to be fully compatible with the Cardano ecosystem, supporting the Cardano-node and Cardano wallet integration.
- This makes it easier to deploy and interact with smart contracts on the Cardano blockchain.

7. Low Learning Curve for Developers:

- Aiken is intended to have a lower learning curve for developers, especially those familiar with functional programming languages.
- Its design philosophy is to be intuitive and accessible, which helps in broadening the developer base for Cardano.

8. Efficient Resource Usage:

- Contracts written in Aiken are designed to be resource-efficient, optimizing the use of Cardano's UTXO model and minimizing transaction fees.
- This is crucial for maintaining the scalability and sustainability of the Cardano network.

9. Formal Verification Support:

- Aiken supports formal verification, allowing developers to prove the correctness of their smart contracts mathematically.
- This feature enhances the reliability and security of contracts, especially in critical applications like finance.

10. Open Source and Community-Driven:

- Aiken is open-source, encouraging contributions and improvements from the Cardano community.
- The language evolves through community feedback and collaboration, ensuring that it meets the needs of developers.

These features make Aiken an attractive choice for developers looking to create smart contracts on Cardano, offering a balance of usability, performance, and security.

4. List of new features what we can add to aiken

To enhance Aiken as a smart contract language for the Cardano blockchain, here are some potential new features that could be considered for development. These additions would aim to improve the developer experience, increase security, and expand the language's capabilities:

1. Enhanced Error Handling and Debugging Tools

- **Feature**: Introduce more sophisticated error handling constructs, such as try-catch blocks, to better manage exceptions and failures within smart contracts.
- **Benefit**: Provides clearer insights into contract failures, helping developers quickly diagnose and fix issues.

2. Integrated Contract Analytics and Performance Profiling

- **Feature**: Built-in tools to analyze the performance of smart contracts, including gas/transaction fee estimations and execution time analysis.
- **Benefit**: Allows developers to optimize contracts for performance and cost-efficiency before deployment.

3. Extended Cryptographic Functions

- **Feature**: Add more cryptographic utilities, such as multi-signature schemes, zero-knowledge proofs, and advanced hashing algorithms.
- **Benefit**: Enhances security and enables more complex contract designs, particularly for DeFi applications.

4. Smart Contract Modularization and Reusability

- **Feature**: Support for modular and reusable contract components or libraries, allowing developers to easily share and reuse code across multiple contracts.
- **Benefit**: Reduces code duplication, speeds up development, and promotes best practices across the community.

5. Native Support for Cross-Chain Interoperability

- **Feature**: Implement primitives that facilitate cross-chain interactions, allowing contracts on Cardano to interact with contracts on other blockchains like Ethereum or Bitcoin.
- **Benefit**: Expands the potential use cases for Cardano contracts, enabling a more connected blockchain ecosystem.

6. Contract State Management and Lifecycle Hooks

- **Feature**: Introduce state management constructs that allow developers to better control the lifecycle of contracts, including initialization, state transitions, and cleanup.
- **Benefit**: Makes it easier to manage complex stateful applications, such as games or complex financial protocols.

7. Improved Formal Verification Support

- **Feature**: Integration with formal verification tools that can automatically generate proofs of contract correctness, ensuring the contract behaves as intended.
- **Benefit**: Increases the reliability and security of contracts, particularly for high-stakes applications like financial systems.

8. Enhanced User Interaction with Smart Contracts

- **Feature**: New constructs that allow for more dynamic user interactions, such as improved input validation and feedback mechanisms within contracts.
- **Benefit**: Makes contracts more user-friendly and can enhance the UX of decentralized applications.

9. AI-Assisted Code Suggestions and Contract Optimization

- **Feature**: Integrate AI-based tools that provide code suggestions, identify potential vulnerabilities, and suggest optimizations for gas efficiency.
- **Benefit**: Helps developers write better code faster and reduces the likelihood of errors.

10. Graphical Contract Development Environment

- **Feature**: A visual IDE extension or plugin that allows developers to create and test contracts using a drag-and-drop interface, similar to visual programming tools.
- **Benefit**: Lowers the entry barrier for non-programmers and speeds up the prototyping process for developers.

11. Support for Oracles and External Data Feeds

- **Feature**: Native support for oracles that bring real-world data into contracts, enabling contracts to react to external events (e.g., price feeds, weather data).
- **Benefit**: Expands the functionality of smart contracts, making them more versatile for a wide range of applications.

12. Composable Smart Contracts and Function Chaining

- **Feature**: Allow contracts to easily call functions from other contracts and compose multiple contracts into a single, seamless execution flow.
- **Benefit**: Facilitates complex workflows and interactions between different smart contracts, enabling richer decentralized applications.

These new features would significantly enhance Aiken's capabilities, making it a more powerful and versatile language for developing Cardano smart contracts while improving security, usability, and overall developer experience.

5. New features testing results
6. And the already developed features must have testing phase demonstrating reliability efficiency and compatibility with details based on cardano ecosystem
7. Issued identification and solving

LIST OF ALL NEW DEVELOPMENT FUNCTIONALITY OF AIKEN-LANG

-
-
-

LIST OF NEW FEATURES DEVELOPED IN AND CAN BE DEVELOPED IN AIKEN-LANG

1.
2.
3.

Aiken vs plutus

Aiken and Plutus are both languages designed for writing smart contracts on the Cardano blockchain, but they have distinct differences in terms of design philosophy, usability, performance, and developer experience. Below is a comparison of Aiken and Plutus to help you understand their key differences, advantages, and use cases:

1. Language Design and Purpose

- **Aiken:**
  - Aiken is a new smart contract language specifically designed for the Cardano blockchain, focusing on improving the developer experience, performance, and safety of smart contract development.
  - It aims to provide a simpler, more accessible, and user-friendly approach compared to Plutus, especially for developers who may not be familiar with Haskell.
  - Aiken's syntax is more approachable, with a focus on readability and ease of use.
- **Plutus:**
  - Plutus is Cardano's original smart contract platform based on Haskell, a functional programming language known for its strong type safety and expressive power.
  - Plutus allows developers to write smart contracts using Haskell, which compiles to Plutus Core, the low-level language that runs on the Cardano blockchain.
  - Plutus was designed to leverage Haskell's robust type system and formal verification capabilities, making it powerful but with a steep learning curve.

2. Developer Experience

- **Aiken:**
  - Aiken focuses on improving the developer experience with a simpler syntax and faster compilation times. It's designed to be more accessible for developers who are not Haskell experts.
  - Aiken's tooling is lightweight, emphasizing a smooth development workflow with better error messages, rapid testing, and more straightforward debugging.
  - It offers a more modern and familiar development environment for developers coming from other languages, making it easier to adopt.
- **Plutus:**
  - Plutus, being based on Haskell, is powerful but has a high barrier to entry, especially for developers who are not familiar with functional programming paradigms.
  - The Plutus development environment can be complex, requiring knowledge of Haskell and advanced programming concepts such as monads, algebraic data types, and functional design patterns.
  - Compilation times in Plutus can be slower, and the error messages can be more challenging to interpret, making debugging more difficult.

3. Performance and Compilation

- **Aiken:**
  - Aiken is designed to compile directly to Plutus Core, optimized specifically for Cardano's Extended UTXO (EUTXO) model. This direct compilation improves performance and reduces overhead.
  - The language focuses on efficiency and aims to minimize gas costs by generating highly optimized bytecode, which is crucial for cost-sensitive smart contracts.
  - Faster compilation times and a more efficient runtime are major performance advantages.
- **Plutus:**
  - Plutus contracts also compile to Plutus Core, but due to the complexity of Haskell, they can sometimes produce less optimized code, especially if not written carefully by experienced developers.
  - The performance of Plutus smart contracts can vary significantly depending on the developer's familiarity with Haskell's optimization techniques.
  - Slower compilation times compared to Aiken can make the development cycle longer.

4. Ecosystem and Tooling

- **Aiken:**
  - Aiken is still developing its ecosystem, but it is designed to work seamlessly with Cardano's existing tools and libraries.
  - The focus is on building a suite of easy-to-use, developer-friendly tools, such as testing frameworks and static analyzers tailored for Aiken's syntax and approach.
- **Plutus:**
  - Plutus has a more mature ecosystem with a variety of tools, libraries, and community support, including the Plutus Application Backend (PAB) for managing contract interactions.
  - However, the tooling is more complex and often requires a deeper understanding of Haskell and

the Cardano architecture.

5. Learning Curve

- **Aiken:**
  - Aiken is designed to be more accessible, with a shallower learning curve compared to Plutus. It targets developers who may be new to Cardano or smart contract development.
  - The syntax and design prioritize readability, making it easier for developers from other programming backgrounds to pick up quickly.
- **Plutus:**
  - Plutus has a steep learning curve, especially for those not familiar with Haskell or functional programming.
  - Mastery of Plutus requires a deep understanding of Haskell's advanced features, which can be a barrier for new developers.

6. Use Cases and Target Audience

- **Aiken:**
  - Ideal for developers who want to quickly and efficiently build smart contracts on Cardano without delving into the complexities of Haskell.
  - Best suited for small to medium-sized projects where rapid development, readability, and ease of maintenance are critical.
- **Plutus:**
  - Best suited for projects that require highly complex, formally verified smart contracts where the full power of Haskell's type system can be leveraged.
  - Suitable for developers who are already comfortable with Haskell or those building highly specialized financial contracts.

7. Community and Support

- **Aiken:**
  - A growing community with a focus on simplifying Cardano smart contract development. As it's newer, the community is still expanding.
  - Offers the potential to become the go-to choice for developers seeking simplicity and ease of use on Cardano.
- **Plutus:**
  - A well-established community with extensive resources, documentation, and experienced developers available to offer support.
  - A broader base of existing projects and tools available, though often with a higher complexity level.

Summary

- **Aiken** is about simplicity, performance, and a better developer experience, making it ideal for developers new to Cardano or looking for a faster, more accessible way to write smart contracts.
- **Plutus** is about power, precision, and deep integration with Cardano's EUTXO model, but it requires significant expertise in Haskell and functional programming.

Aiken can be seen as a more developer-friendly alternative that aims to lower barriers to entry while maintaining performance, making it a compelling choice for a broad audience looking to build on Cardano.

Aiken-lang smart contract basic features in cardano ecosystem

INTRODUCTION

Aiken-lang is a domain-specific language designed for writing smart contracts on the Cardano blockchain. It aims to simplify and enhance the development process while leveraging Cardano's unique capabilities.

KEY FEATURES OF AIKEN-LANG

1. **Simplicity and Readability**
   - Aiken-lang promotes a syntax that is easy to understand, reducing the barrier to entry for developers.
   - Its language constructs are designed to be clear, which assists in writing and reviewing smart contracts efficiently.
2. **Type Safety**
   - Aiken-lang utilizes strong static typing, which allows developers to catch errors at compile time rather than runtime.
   - The type system helps ensure that smart contracts behave as intended, enhancing security and reliability.
3. **Integration with Plutus**
   - Aiken-lang is built to work seamlessly with Cardano's Plutus framework, which supports the creation of complex financial applications and contracts.
   - It allows developers to utilize Plutus features and libraries while writing contracts in a more accessible syntax.
4. **Modular Design**
   - The language supports modular programming paradigms, enabling developers to organize code into reusable components.
   - This modularity facilitates easier maintenance and upgrades of smart contracts.
5. **Testing and Simulation Tools**
   - Aiken-lang provides built-in testing frameworks to simulate smart contracts before deployment.
   - Developers can create test scenarios that mimic real-world interactions, thereby ensuring contract functionality and security.
6. **Interoperability with Cardano Ecosystem**
   - Aiken-lang is designed to work natively within the Cardano ecosystem, allowing for interaction with other Cardano-based applications and services.
   - Its principles are aligned with Cardano's philosophy of sustainability, scalability, and interoperability.

7. **Advanced Features for Functional Programming**
   - Aiken-lang supports functional programming paradigms, allowing developers to leverage higher-order functions and immutability.
   - This approach enhances the expressiveness of contracts and leads to more robust code.
8. **Efficient Resource Management**
   - L The language incorporates resource management features that help developers write contracts optimizing transaction costs and network resources.
   - L It encourages practices that minimize gas fees and maximize efficiency.
9. **Community and Ecosystem Support**
   - Aiken-lang is supported by a growing community of developers who contribute to its libraries and tools.
   - As part of the Cardano ecosystem, it benefits from a wealth of resources, documentation, and collaboration opportunities.
10. **I L Documentation and Learning Resources**
   - Comprehensive documentation is provided, which includes tutorials, guides, and best practices for writing smart contracts.
   - This educational emphasis aids developers at all skill levels in effectively utilizing the language.

CONCLUSION

Aiken-lang stands out in the Cardano ecosystem with its user-friendly, type-safe, and modular design, empowering developers to create reliable and efficient smart contracts. Its integration with the Cardano blockchain and Plutus framework, along with support for functional programming principles, positions Aiken-lang as a prominent choice for developers seeking to leverage smart contracts in decentralized applications. As the ecosystem continues to evolve, Aiken-lang is likely to play a critical role in the advancement of Cardano's smart contract capabilities.

Testing simple Aiken smart contract

Testing Aiken contracts is crucial for ensuring the correctness and reliability of your smart contracts before deploying them on the Cardano blockchain. Aiken-lang provides tools and practices to facilitate testing, which involves unit testing individual functions, simulating transactions, and validating contract behavior in various scenarios.

Here’s a step-by-step guide to testing Aiken contracts:

1. Setting Up a Test Environment

Before you start writing tests, ensure you have the following:

- **Aiken CLI**: Aiken provides a command-line interface (CLI) for compiling, testing, and deploying contracts. Ensure you have it installed and set up.
- **Test Framework**: Aiken might include a testing framework within its ecosystem, or you might need to set up a simple testing script to execute and validate outputs.

2. Writing Unit Tests

Unit tests are designed to test individual functions in isolation. This is the most granular level of testing and should cover all possible input cases.

EXAMPLE UNIT TEST IN AIKEN

```
module ExampleTest

import aiken/assert
import Example

# Test a simple validator
test_validate_matching_datum_redeemer {
    let datum = ByteArray.fromString("test")
    let redeemer = ByteArray.fromString("test")
    let context = ScriptContext.mock

    let result = Example.validator(datum, redeemer, context)

    assert.equal(result, true)
}

# Test a failure case
test_validate_non_matching_datum_redeemer {
    let datum = ByteArray.fromString("test1")
    let redeemer = ByteArray.fromString("test2")
    let context = ScriptContext.mock

    let result = Example.validator(datum, redeemer, context)

    assert.equal(result, false)
}
```

EXPLANATION:

- `test_validate_matching_datum_redeemer` : Tests a scenario where the `datum` matches the `redeemer` , expecting a `true` result.
- `test_validate_non_matching_datum_redeemer` : Tests the opposite scenario, where `datum` and `redeemer` don't match, expecting a `false` result.

3. Mocking Blockchain Contexts

Aiken contracts typically interact with the blockchain context (`ScriptContext`), which includes transaction information. When testing, you need to mock these contexts to simulate different scenarios.

EXAMPLE OF MOCKING CONTEXT

```
module ExampleTest

import aiken/assert
import Example
import aiken/context

# Mocked context for a specific test case
mock_script_context = {
    let inputs = [ /* Mock transaction inputs */ ]
    let outputs = [ /* Mock transaction outputs */ ]
    let fee = 2000
    let tx_info = context.TxInfo { inputs, outputs, fee, ... }

    ScriptContext { tx_info }
}

# Use the mocked context in a test
test_validate_with_mocked_context {
    let datum = ByteArray.fromString("test")
    let redeemer = ByteArray.fromString("test")
    let context = mock_script_context

    let result = Example.validator(datum, redeemer, context)

    assert.equal(result, true)
}
```

EXPLANATION:

- `mock_script_context` : Creates a mock `ScriptContext` with specific transaction inputs, outputs, and fee.
- `test_validate_with_mocked_context` : Tests the validator function using this mocked context.

4. Running Tests

You can run your tests using the Aiken CLI:

```
aiken test
```

This command will execute all tests in your module and report the results, indicating which tests passed or failed.

5. Simulating Full Transactions

Beyond unit tests, you can simulate full transactions to test how your contract behaves in a real-world scenario. This involves creating a full transaction mock, including all inputs, outputs, and other relevant data, and then running your contract logic against it.

EXAMPLE OF SIMULATING A TRANSACTION

```
module FullTransactionTest

import aiken/assert
import Example
import aiken/context

test_simulate_transaction {
    let inputs = [ /* Full mock inputs */ ]
    let outputs = [ /* Full mock outputs */ ]
    let fee = 1000
    let tx_info = context.TxInfo { inputs, outputs, fee, ... }
    let context = ScriptContext { tx_info }

    let datum = ByteArray.fromString("test")
    let redeemer = ByteArray.fromString("test")

    let result = Example.validator(datum, redeemer, context)

    assert.equal(result, true)
}
```

EXPLANATION:

- This simulates a full transaction, using all relevant parts of the `ScriptContext`.
- Tests are run to ensure the validator function behaves as expected in this complex scenario.

6. Edge Cases and Failure Scenarios

Ensure you test edge cases, such as:

- Extremely high or low values.
- Missing or incorrect transaction details.
- Unexpected inputs (e.g., wrong types or formats).

7. Debugging Test Failures

When a test fails:

- Review the error message and the exact test case that failed.
- Use `trace` statements in your code to log intermediate values.
- Modify the test or the contract code as necessary to handle the scenario properly.

8. Test Coverage

Finally, ensure you have comprehensive test coverage. Every function, condition, and possible interaction should be tested. This reduces the likelihood of bugs making it into the deployed contract.

Conclusion

Testing Aiken contracts is a systematic process that involves unit testing, mocking blockchain contexts, simulating transactions, and running comprehensive test cases to cover all possible scenarios. By rigorously testing your contracts, you can deploy them with greater confidence, knowing they will behave correctly on the Cardano blockchain.

Debug aiken errors ?

Debugging errors in Aiken-lang involves identifying and understanding common error messages and tracing them back to the source in your code. Since Aiken-lang is a relatively new language designed for writing smart contracts on Cardano, the debugging process often involves working directly with Aiken's syntax and semantics, as well as understanding the underlying blockchain context.

Here's a guide to help you debug common Aiken-lang errors:

Common Aiken Error Categories

1. **Syntax Errors**: Issues with the code syntax, such as missing semicolons, incorrect use of keywords, or mismatched parentheses.
2. **Type Errors**: Incorrect types being used, like passing an integer where a string is expected.
3. **Runtime Errors**: Errors that occur during execution, such as division by zero or unhandled conditions.
4. **Logical Errors**: Errors in the logic of your code, which cause it to behave differently than expected.
5. **Contextual Errors**: Errors related to the blockchain context, such as incorrect use of `ScriptContext` or invalid transaction details.

Step-by-Step Debugging Process

1. **Identify the Error Message:** Aiken will usually provide an error message when something goes wrong. Read the error message carefully, as it often points directly to the line or function causing the issue.
2. **Check Syntax and Structure**:
   - Ensure all syntax rules are followed, like proper indentation, use of semicolons, and correct function declarations.
   - Verify that all imports are correctly referenced and the module is properly defined.
3. **Validate Types**:
   - Ensure that the types of inputs and outputs match what the functions expect.
   - Check the function signatures and compare them with the actual arguments being passed.

4. **Use Trace Statements**:
   - Use `trace` functions to output intermediate values and understand how data flows through your code. This is especially useful for pinpointing where things go wrong.
     trace "Starting validation..." (validator datum redeemer context)

```
```

5. **Error Handling**:
   - Use clear error messages with the `error` function to know exactly why a function failed.
     if condition_failed {
       error("Condition failed due to mismatched datum and redeemer")
     }

```
```

6. **Examine Script Context**:
   - Check the `ScriptContext` carefully to ensure you are accessing the correct fields.
   - Make sure you are using the correct transaction details, like inputs, outputs, and signatures.
7. **Check Built-in Functions**:
   - Verify that built-in functions (e.g., `verify_signature` , `member` , `contains`) are being used correctly and receiving the right types of arguments.
8. **Simulate Execution**:
   - Use test environments or simulators that Aiken provides to run your contract in a controlled setting, which can help identify runtime errors before deploying on the blockchain.

Debugging Examples

EXAMPLE 1: TYPE ERROR

Error Message:

```
Type mismatch: expected Bool but got Int
```

Solution:

- Check where a `Bool` was expected (e.g., in a conditional), but an `Int` was provided.
- Ensure you are returning the correct type.

EXAMPLE 2: INVALID CONTEXT ACCESS

Error Message:

```
Invalid access to context field: tx_info.fee
```

Solution:

- Double-check that you are accessing `tx_info` correctly.
- Make sure the structure of `ScriptContext` is correctly navigated.

EXAMPLE 3: UNEXPECTED ERROR TRIGGER

Error Message:

```
Validation failed: Custom error triggered
```

Solution:

- Look at all `error` calls in the code to trace which condition triggers it.
- Use `trace` to log intermediate states just before the error occurs.

Best Practices for Debugging Aiken

1. **Write Test Cases**: Use unit tests to validate your functions individually. Isolate functions to confirm they behave as expected.
2. **Keep Code Modular**: Break down your code into small, testable functions. This makes it easier to find and fix bugs.
3. **Review Documentation**: Reference Aiken-lang documentation frequently to ensure correct use of language features.
4. **Community and Tools**: Engage with the Aiken community, such as forums or Discord, where you can ask specific questions or share code snippets for feedback.

Debugging in Aiken-lang can be streamlined by following these steps, utilizing trace functions, and carefully reviewing the code structure and context interactions.

Aiken Deployment process

Deploying an Aiken smart contract to the Cardano blockchain involves several steps, including compiling the contract, preparing the necessary scripts and transaction files, and submitting the contract through a Cardano node. Below is a detailed guide to help you deploy your Aiken contract:

Step-by-Step Guide to Deploying an Aiken Contract

1. Prerequisites

Before deploying, ensure you have the following set up:

- **Aiken CLI**: Installed and properly configured. This tool is necessary for compiling and interacting with Aiken contracts.
- **Cardano Node**: Running a synced Cardano node is essential for interacting with the blockchain.
- **Cardano CLI**: The Cardano command-line interface is required for creating and submitting transactions.
- **Cardano Wallet**: A wallet with sufficient ADA for fees and collateral, such as Daedalus, Yoroi, or any command-line wallet compatible with Cardano.

2. Compile Your Aiken Contract

Compile your Aiken smart contract using the Aiken CLI. The compilation step converts your Aiken code into a format that can be deployed on the blockchain (Plutus scripts).

```
aiken build
```

This command will compile your Aiken source code and generate the necessary Plutus script files (often in JSON format) in the `output` directory.

3. Prepare the Plutus Script

The output of the compilation will include a JSON file containing your Plutus script. Make sure you know the path to this file, as you'll need it for the deployment.

For example, if your contract is called `Example` , the compiled script might be saved as:

```
output/Example.plutus
```

4. Create a Datum, Redeemer, and Script Context

To interact with the script, you need to define the **datum** (data attached to the script), **redeemer** (data used to unlock the script), and other contextual details.

- **Datum**: The input data for your script, often a JSON file or a simple value.
- **Redeemer**: The data used during execution to validate conditions, also in JSON or simple format.

Example files:

- `datum.json`
- `redeemer.json`

5. Set Up the Transaction

You need to set up a transaction that includes the Plutus script, which usually involves locking some ADA or tokens at the script address. Follow these steps:

1. **Generate the Script Address**: Use the Cardano CLI to generate the address for your script.
   cardano-cli address build --payment-script-file output/Example.plutus --mainnet

   Save the generated address as it will be used to send ADA or other tokens.
2. **Fund the Script Address**: Send ADA or tokens to the script address from your wallet. You will need the UTxO (Unspent Transaction Output) from this step for the actual deployment transaction.
3. **Create the Transaction**: Create a transaction that interacts with the Plutus script. This involves:
   - Specifying the input UTxO.
   - Defining the output (e.g., ADA sent to the script address).
   - Including the datum, redeemer, and any additional required script context.

Example command:

```
cardano-cli transaction build \
    --alonzo-era \
    --tx-in <UTxO> \
    --tx-out <script-address>+<amount> \
    --tx-out-datum-hash-file datum.json \
    --protocol-params-file protocol.json \
    --out-file tx.raw
```

6. Sign the Transaction

Sign the raw transaction file with your wallet's signing key:

```
cardano-cli transaction sign \
    --signing-key-file payment.skey \
    --tx-body-file tx.raw \
    --out-file tx.signed
```

7. Submit the Transaction

Submit the signed transaction to the blockchain:

```
cardano-cli transaction submit --tx-file tx.signed --mainnet
```

8. Verify the Deployment

After submitting, verify that the transaction was successful and the contract is deployed:

- **Check the UTxO**: Use `cardano-cli query utxo` to check the UTxOs at your script address.
  cardano-cli query utxo --address <script-address> --mainnet

```

```

- **Interact with the Contract**: Test interactions by executing transactions that use the redeemer to unlock funds or trigger contract behavior.

9. Debugging Deployment Issues

If the deployment fails, consider the following steps:

- **Check Transaction Fees**: Ensure sufficient fees are included.
- **Inspect Errors**: If the transaction fails during script execution, review the error logs to understand the issue (e.g., datum/redeemer mismatches).
- **Adjust Parameters**: Verify the correctness of datum, redeemer, and script context details.

Summary

Deploying an Aiken contract involves compiling the code, setting up transaction details, signing, and submitting the transaction to the blockchain. The key is to ensure that all components (datum, redeemer, and context) are correctly defined and that sufficient funds are available for fees. Careful preparation and testing in a testnet environment can help mitigate issues during the mainnet deployment.

Aiken-lang official functions

- [ ] Aiken-lang official features
- [ ] Aiken-lang features issues
- [ ] Aiken-lang features issue solutions
- [ ] Debug aiken errors?

Debugging errors in Aiken-lang involves identifying and understanding common error messages and tracing them back to the source in your code. Since Aiken-lang is a relatively new language designed for writing smart contracts on Cardano, the debugging process often involves working directly with Aiken's syntax and semantics, as well as understanding the underlying blockchain context.

Here's a guide to help you debug common Aiken-lang errors:

Common Aiken Error Categories

1. **Syntax Errors**: Issues with the code syntax, such as missing semicolons, incorrect use of keywords, or mismatched parentheses.
2. **Type Errors**: Incorrect types being used, like passing an integer where a string is expected.
3. **Runtime Errors**: Errors that occur during execution, such as division by zero or unhandled conditions.
4. **Logical Errors**: Errors in the logic of your code, which cause it to behave differently than expected.
5. **Contextual Errors**: Errors related to the blockchain context, such as incorrect use of `ScriptContext` or invalid transaction details.

Step-by-Step Debugging Process

1. **Identify the Error Message**: Aiken will usually provide an error message when something goes wrong. Read the error message carefully, as it often points directly to the line or function causing

the issue.

2. **Check Syntax and Structure**:
   - Ensure all syntax rules are followed, like proper indentation, use of semicolons, and correct function declarations.
   - Verify that all imports are correctly referenced and the module is properly defined.
3. **Validate Types**:
   - Ensure that the types of inputs and outputs match what the functions expect.
   - Check the function signatures and compare them with the actual arguments being passed.
4. **Use Trace Statements**:
   - Use `trace` functions to output intermediate values and understand how data flows through your code. This is especially useful for pinpointing where things go wrong.

     trace "Starting validation..." (validator datum redeemer context)

```
```

5. **Error Handling**:
   - Use clear error messages with the `error` function to know exactly why a function failed.

     if condition_failed {

       error("Condition failed due to mismatched datum and redeemer")

     }

```
```

6. **Examine Script Context**:
   - Check the `ScriptContext` carefully to ensure you are accessing the correct fields.
   - Make sure you are using the correct transaction details, like inputs, outputs, and signatures.
7. **Check Built-in Functions**:
   - Verify that built-in functions (e.g., `verify_signature` , `member` , `contains`) are being used correctly and receiving the right types of arguments.
8. **Simulate Execution**:
   - Use test environments or simulators that Aiken provides to run your contract in a controlled setting, which can help identify runtime errors before deploying on the blockchain.

Debugging Examples

EXAMPLE 1: TYPE ERROR

Error Message:

```
Type mismatch: expected Bool but got Int
```

Solution:

- Check where a `Bool` was expected (e.g., in a conditional), but an `Int` was provided.
- Ensure you are returning the correct type.

EXAMPLE 2: INVALID CONTEXT ACCESS

Error Message:

```
Invalid access to context field: tx_info.fee
```

Solution:

- Double-check that you are accessing `tx_info` correctly.
- Make sure the structure of `ScriptContext` is correctly navigated.

EXAMPLE 3: UNEXPECTED ERROR TRIGGER

Error Message:

```
Validation failed: Custom error triggered
```

Solution:

- Look at all `error` calls in the code to trace which condition triggers it.
- Use `trace` to log intermediate states just before the error occurs.

Best Practices for Debugging Aiken

1. **Write Test Cases**: Use unit tests to validate your functions individually. Isolate functions to confirm they behave as expected.
2. **Keep Code Modular**: Break down your code into small, testable functions. This makes it easier to find and fix bugs.
3. **Review Documentation**: Reference Aiken-lang documentation frequently to ensure correct use of language features.
4. **Community and Tools**: Engage with the Aiken community, such as forums or Discord, where you can ask specific questions or share code snippets for feedback.

Debugging in Aiken-lang can be streamlined by following these steps, utilizing trace functions, and carefully reviewing the code structure and context interactions.

Fix aiken-lang common errors

Fixing common errors during the development and deployment of Aiken smart contracts involves understanding typical issues that can occur at various stages—compilation, testing, transaction creation, signing, and execution. Here is a guide to help you identify, troubleshoot, and fix some of the most common errors encountered when working with Aiken contracts:

Common Errors in Aiken Contracts and How to Fix Them

1. Compilation Errors

Error Message:

```
Syntax error: unexpected token
```

Cause and Fix:

- This is usually due to typos, missing or extra punctuation, incorrect indentation, or malformed syntax.
- **Fix:** Double-check the line number mentioned in the error message. Ensure that all keywords, braces, and delimiters are correctly placed. Review the function signatures and control structures.

Error Message:

```
Type mismatch: expected <type> but got <different type>
```

Cause and Fix:

- This occurs when the function receives an argument of an incorrect type, such as passing an `Int` where a `Bool` is expected.
- **Fix:** Check the function signature and ensure the arguments match the expected types. Use type casting functions or refactor the function to accept the correct types.

Error Message:

```
Unknown module or function
```

Cause and Fix:

- The contract is attempting to import a module or call a function that does not exist or is incorrectly referenced.
- **Fix:** Verify the module path and ensure the imported function is correctly spelled. Make sure the module is part of the correct project structure and is included in the build configuration.

2. Testing Errors

Error Message:

```
Assertion failed: expected true but got false
```

Cause and Fix:

- This indicates that a test assertion failed, usually because the function did not return the expected result.
- **Fix:** Add `trace` statements to log intermediate values and review the function logic to ensure it handles all cases correctly. Adjust the function or the test inputs as necessary.

Error Message:

```
ScriptContext mismatch or missing fields
```

Cause and Fix:

- The test is using an incorrectly mocked `ScriptContext` , missing required fields, or using incorrect values.
- **Fix:** Ensure the `ScriptContext` mock matches the expected structure of the actual blockchain context. Check that all relevant transaction details, such as inputs, outputs, and fees, are properly defined.

3. Transaction Creation Errors

Error Message:

```
Insufficient funds for transaction fee
```

Cause and Fix:

- The transaction does not have enough ADA to cover the required fees.
- **Fix:** Check your wallet balance and ensure sufficient ADA is available. You may need to adjust the

transaction outputs or increase the funds being sent.

Error Message:

```
Invalid datum or redeemer format
```

Cause and Fix:

- The datum or redeemer is incorrectly formatted or does not match the expected type.
- **Fix:** Ensure that the datum and redeemer are correctly formatted JSON files or byte arrays, as required by the contract. Validate that they conform to the expected structure.

Error Message:

```
Invalid script address
```

Cause and Fix:

- This occurs when the provided script address is malformed or incorrectly generated.
- **Fix:** Regenerate the script address using the correct Plutus script file and Cardano CLI command. Verify that the address is correctly formatted and corresponds to the correct network (mainnet or testnet).

4. Signing Errors

Error Message:

```
Invalid signing key or missing signature
```

Cause and Fix:

- This happens when the signing key is incorrect, missing, or the transaction does not include the necessary signatures.
- **Fix:** Ensure that the correct signing key file (`.skey`) is used and matches the address from which funds are being sent. Re-sign the transaction with the correct key.

Error Message:

```
Transaction signing failed: Bad input
```

Cause and Fix:

- The transaction has invalid inputs or is missing required components.
- **Fix:** Rebuild the transaction, ensuring that all inputs are valid UTxOs and that all necessary files (datum, redeemer, and script context) are included.

5. Execution Errors on the Blockchain

Error Message:

```
Validation script failed: Exceeded execution units
```

Cause and Fix:

- The script used more computational resources than allowed.
- **Fix:** Optimize the contract code to reduce resource consumption. Check loops and complex calculations. You may need to increase the `execution units` limit in your transaction settings if permissible.

Error Message:

```
Script failed due to redeemer error: Custom error
```

Cause and Fix:

- The contract logic failed due to an incorrect redeemer or a condition that wasn't met.
- **Fix:** Double-check the redeemer data and ensure it meets the conditions expected by the contract. Add more detailed `trace` statements to log where the failure occurs within the script.

6. General Debugging Tips

- **Use Trace Statements**: Insert `trace` statements throughout your contract code to log intermediate values and conditions. This will help identify exactly where the error occurs.
- **Test Incrementally**: Start with simple tests and gradually add complexity. This helps isolate where problems occur as you add new functionality.

- **Simulate Transactions**: Before deploying on the mainnet, use the Cardano testnet to simulate transactions, allowing you to test without risking real funds.
- **Check Logs**: Cardano node and CLI logs often provide additional error details that are not immediately obvious from the CLI output.
- **Community Support**: Engage with the Aiken and Cardano communities through forums or Discord channels. Sharing specific error messages can help you get targeted advice.

Fixing errors in Aiken contracts involves careful examination of the error messages, iterative testing, and ensuring that all components are correctly formatted and configured. By systematically troubleshooting and following best practices, you can resolve most common issues encountered during development and deployment.

Audit tools for Aiken-lang

As of now, Aiken-lang is relatively new, and the ecosystem around it is still developing. Consequently, dedicated audit tools specifically designed for Aiken-lang may not yet be as mature or widely available as those for more established smart contract languages like Solidity. However, given its design and compatibility with the Cardano blockchain, developers can use several approaches and tools to audit Aiken-lang smart contracts effectively.

1. Formal Verification Tools

- **Coq / Agda:** These are general-purpose proof assistants that can be used for formal verification. While not specific to Aiken, they can help you prove properties about your smart contracts if you translate your Aiken logic into a formal language that these tools understand.

2. Manual Code Review

- **Peer Reviews:** Engage in thorough manual code reviews with experienced developers. Given the strong typing and clear syntax of Aiken, this can be highly effective in identifying potential issues.

3. Static Analysis Tools

- **Plutus Static Analysis Tools:** Since Aiken compiles to Plutus Core, you can use existing Plutus static analysis tools to analyze the compiled output. These tools can help identify common issues such as potential reentrancy, improper handling of UTXOs, and inefficient gas usage.

4. Property-Based Testing

- **QuickCheck for Aiken:** You can adapt property-based testing frameworks like QuickCheck (commonly used in Haskell and related languages) for Aiken. This approach involves defining properties that your contract should satisfy and then automatically generating test cases to verify those properties.

5. Simulation and Testing Frameworks

- **Aiken's Built-in Testing Framework:** Aiken includes its own testing framework, which can be used to write unit tests and simulate various scenarios. By thoroughly testing your contracts under a wide range of conditions, you can catch issues before deployment.

6. Formal Audits by Third-Party Security Firms

- **Blockchain Security Firms:** Engaging a security firm that specializes in Cardano or Plutus-based contracts can be a good option. These firms can adapt their existing tools and methodologies to audit Aiken contracts effectively.

7. Manual Symbolic Execution

- **Symbolic Execution with Manual Tools:** While automated symbolic execution tools for Aiken may not be available, you can manually perform symbolic execution, where you track the flow of symbolic inputs through your contract to identify potential vulnerabilities.

8. Community Tools and Scripts

- **Custom Scripts:** Given the newness of Aiken, developers may need to write custom scripts to perform certain checks or validations. Sharing these tools within the community can help build a repository of resources for Aiken auditing.

9. Continuous Integration (CI) Pipelines

- **CI/CD Integration:** Integrate your Aiken contracts with CI/CD pipelines that automatically run tests and checks whenever you update your code. This can help ensure that your contracts are continuously vetted for security issues.

10. Monitoring and On-Chain Analysis Tools

- **Cardano-Specific Monitoring:** Once deployed, use on-chain monitoring tools to track your contract's behavior in the wild. Tools that monitor Cardano transactions can be adapted to keep an eye on the performance and security of your Aiken contracts.

Future Developments

As Aiken matures, more specialized audit tools are likely to emerge. Keeping an eye on the Aiken and Cardano developer communities, contributing to open-source efforts, and adapting existing tools can help fill the current gaps in Aiken-specific auditing capabilities.

Aiken lang best security partices

When developing smart contracts with Aiken-lang, a language specifically designed for Cardano's blockchain, it's important to follow best security practices to protect against vulnerabilities and ensure that your contracts function as intended. Here are the best security practices tailored for Aiken-lang:

1. Leverage Strong Typing for Safety

- **Type-Safe Operations:** Utilize Aiken's strong typing system to enforce type safety, reducing the risk of common programming errors like unintended data manipulation or type mismatches.
- **Explicitness in Types:** Be explicit with your types to prevent unexpected behaviors. Avoid using overly generic types that could introduce ambiguity.

2. Use Formal Verification Tools

- **Formal Verification:** Make use of Aiken's formal verification capabilities to mathematically prove the correctness of your contract logic. This process helps identify and eliminate logic errors before deployment.

3. Keep Contracts Modular and Simple

- **Single Responsibility Principle:** Design your contracts so that each module or function has a single, clear responsibility. This reduces complexity and makes the contract easier to audit and test.
- **Reuse Proven Code:** Reuse existing, well-audited libraries and modules where possible. Avoid reinventing the wheel for standard functions.

4. Implement Comprehensive Testing

- **Unit Testing:** Write unit tests for all critical functions using Aiken's testing framework. Ensure that edge cases and potential attack vectors are covered.
- **Property-Based Testing:** Use property-based testing to explore a wide range of input scenarios, verifying that your contract behaves as expected under various conditions.

5. Practice Defensive Programming

- **Input Validation:** Validate all inputs rigorously. Ensure that inputs are within expected ranges and types before processing.
- **Fail-Safe Mechanisms:** Implement fail-safes in your contract to handle unexpected states or errors gracefully, such as reverting transactions or triggering emergency shutdowns.

6. Apply Proper Access Controls

- **Role-Based Access:** Implement role-based access control mechanisms to restrict critical functions to authorized entities only. Ensure that roles are well-defined and minimal.
- **Multi-Signature Requirements:** For sensitive operations, consider requiring multi-signature approval to reduce the risk of unauthorized actions.

7. Prevent Reentrancy Attacks

- **Check-Effects-Interactions Pattern:** Adopt the check-effects-interactions pattern to avoid reentrancy vulnerabilities. Make state changes before calling external contracts to prevent reentrancy issues.
- **Reentrancy Guards:** Implement reentrancy guards, such as mutexes, to block recursive calls.

8. Optimize Gas Usage

- **Efficient Code Design:** Write code that is optimized for gas efficiency to avoid denial-of-service (DoS) attacks caused by excessive gas consumption.

- **Avoid Expensive Loops:** Minimize the use of loops or recursive operations that could lead to high gas costs.

9. Handle Edge Cases and Errors

- **Fallback Functions:** Implement fallback functions that handle unexpected calls or inputs safely, ensuring that your contract doesn't crash or enter an invalid state.
- **Error Reporting:** Ensure your contract provides meaningful error messages and handles exceptions gracefully.

10. Audit and Peer Review

- **Internal and External Audits:** Regularly audit your smart contracts using both internal reviews and third-party security audits. External auditors can provide a fresh perspective on potential vulnerabilities.
- **Continuous Code Reviews:** Maintain a culture of continuous code reviews within your development team to catch issues early.

11. Implement Upgradability Cautiously

- **Proxy Patterns:** If upgradability is necessary, use well-established proxy patterns with strict access controls. Ensure that upgrades are secure and that the logic behind the contract's critical functions remains intact.
- **Immutable Contracts:** Where possible, prefer immutability for contracts that perform critical functions to prevent unauthorized modifications.

12. Stay Informed on Security Best Practices

- **Keep Up with Developments:** Stay informed about new security threats and best practices in the Cardano and Aiken ecosystems. Regularly update your knowledge and apply new security measures as they evolve.
- **Community Engagement:** Engage with the Aiken development community to share knowledge, learn from others' experiences, and stay updated on best practices.

13. Avoid Using Unverified Libraries

- **Library Audits:** Only use libraries that have been thoroughly audited and are well-documented. Be cautious with new or untested libraries that may introduce vulnerabilities.

14. Gas and Transaction Management

- **Gas Optimization:** Ensure your contract logic is optimized for minimal gas usage, reducing the risk of gas exhaustion during contract execution.
- **Transaction Ordering Awareness:** Be mindful of potential risks related to transaction ordering, especially in environments where front-running or transaction manipulation could occur.

By following these best practices when developing smart contracts with Aiken, you can significantly reduce the risk of vulnerabilities and create robust, secure, and reliable contracts on the Cardano blockchain.

Aiken Official & Unofficial Repo Links

1. Aiken Official Link: https://github.com/aiken-lang/aiken
   Issues: https://github.com/aiken-lang/aiken/issues
   Discussions: https://github.com/aiken-lang/aiken/discussions
2. Aiken official stdlib: https://aiken-lang.github.io/stdlib/
   Source: https://github.com/aiken-lang/stdlib
   Issues: https://github.com/aiken-lang/stdlib/issues
3. Aiken official Functions: https://aiken-lang.org/language-tour/functions#defining-functions
4. Aiken Official Prelude : https://aiken-lang.github.io/prelude/aiken.html
   Source: https://github.com/aiken-lang/prelude
   Issues: https://github.com/aiken-lang/prelude/issues
5. Aiken Official Builtin Functions: https://aiken-lang.github.io/prelude/aiken/builtin.html
6. Aiken official On going Project Tract: https://github.com/aiken-lang/aiken/projects?query=is%3Aopen
7. Aiken Official Documents Site's Code: https://github.com/aiken-lang/site
8. Awesome Aiken : https://github.com/aiken-lang/awesome-aiken
9. The official library for writing fuzzers (a.k.a generators) for the Aiken Cardano smart-contract language.
   Source: https://github.com/aiken-lang/fuzz
   issue: https://github.com/aiken-lang/fuzz/issues
   Testing Repo: https://github.com/avi69night/aiken-lang-fuzz
10. Merkle Patricia Forestry with Aiken
    source: https://github.com/aiken-lang/merkle-patricia-forestry
11. Zed Extensions with aiken
    source: https://github.com/aiken-lang/zed-extensions
12. Zed Aiken: https://github.com/aiken-lang/zed-aiken
13. aiken Homebrew tap: https://github.com/aiken-lang/homebrew-tap
14. Aiken playground: https://play.aiken-lang.org/
    Source: https://github.com/aiken-lang/play
    issue: https://github.com/aiken-lang/play/issues
    leptos: https://github.com/leptos-rs/leptos
15. Aiken Linguist : https://github.com/aiken-lang/linguist
    Aiken forked From: https://github.com/github-linguist/linguist
16. Aiken Tree Sitter Bindings: https://github.com/aiken-lang/tree-sitter-aiken
17. Aikeup- A CLI for multi version of Aiken:
    Source: https://github.com/aiken-lang/aikup
    issue: https://github.com/aiken-lang/aikup/issues
18. sparse-merkle-tree Aiken: https://github.com/aiken-lang/sparse-merkle-tree
19. VsCode Aiken extension
    Source: https://github.com/aiken-lang/vscode-aiken
    issue: https://github.com/aiken-lang/vscode-aiken/issues

20. Aiken Setup: https://github.com/aiken-lang/setup-aiken
21. Emacs mode for Aiken:
    Source: https://github.com/aiken-lang/aiken-mode
22. marlowe - Write validators in the validators folder, and supporting functions in the lib folder using .ak as a file extension.
    Source: https://github.com/aiken-lang/marlowe
23. Aiken Vim - A plugin for working with Aiken on Vim / NeoVim.
    Source: https://github.com/aiken-lang/editor-integration-nvim
24. Aiken-Staking Validator: https://github.com/aiken-lang/staking
    issue: https://github.com/aiken-lang/staking/issues
25. Flat (archive) - This is archived and has been moved to pallas.
    Source: https://github.com/aiken-lang/flat-rs
26. aupic just for fun: https://github.com/aiken-lang/aup lc
27. Aiken escrow-to-in dex-fund : https://github.com/aiken-lang/escrow_to_index_fund
28. Aiken upic-playground - This template should help get you started developing with Tauri, Svelte and TypeScript in Vite.
    Source: https://github.com/aiken-lang/uplc-playground
29. Aiken trees - A library for working with tree data-structures
    Source: https://github.com/aiken-lang/trees
30. Aiken Branding: https://github.com/aiken-lang/branding
31. Aiken Linked List: Linked list structures leverage the EUTXO model to enhancing scalability and throughput significantly. By linking multiple UTXOs together through a series of minting policies and validators, it can improve the user experience interacting with smart contract concurrently.
    Source: https://github.com/Anastasia-Labs/aiken-linked-list

Aiken-lang new features

- [ ] List of Aiken new features can be be developed
- [ ] Testing of new features
- [ ] Test results for new features
- [ ] Errors and issues of new features
- [ ] Solutions of new features issues
- [ ] Demonstrating of all new features
- [ ] Details of new features

Aiken-lang testing with third party

- [ ] Aiken Library for Common Design Patterns in Cardano Smart Contracts (Anastasia-Lab)
- link: https://github.com/Anastasia-Labs/aiken-design-patterns
- To help facilitate faster development of Cardano smart contracts, we present a collection of tried and tested modules and functions for implementing common design patterns.

1. Testing on aiken-lang version
   aiken v1.0.29-alpha+16fb02e
2. Dependency & Pckages:

aiken package add anastasia-labs/aiken-design-patterns --version main

3. Functions of various pattern

use aiken_design_patterns/merkelized_validator as merkelized_validator

use aiken_design_patterns/multi_utxo_indexer as multi_utxo_indexer

use aiken_design_patterns/singular_utxo_indexer as singular_utxo_indexer

use aiken_design_patterns/stake_validator as stake_validator

use aiken_design_patterns/tx_level_minter as tx_level_minter

4. Build Compiling

   Build success

5. aiken design pattern testing results: aiken check

   5.a. aiken_design_patterns/tests

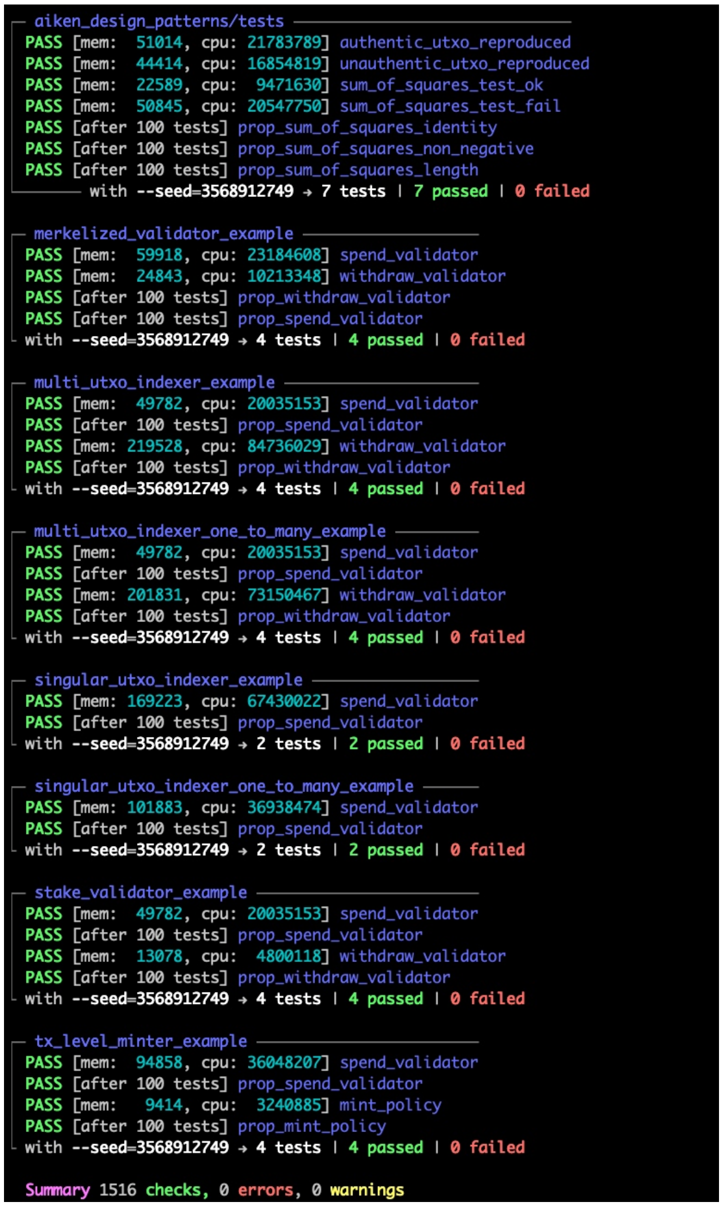
   5.b. merkelized_validator_example

   5.c. multi_utxo_indexer_example

   5.d. singular_utxo_indexer_one_to_many_example

   5.e. stake_validator_example

   5.f. tx_level_minter_example



```
┌ aiken_design_patterns/tests ──────────────────────────────
│ PASS [mem:  51014, cpu: 21783789] authentic_utxo_reproduced
│ PASS [mem:  44414, cpu: 16854819] unauthentic_utxo_reproduced
│ PASS [mem:  22589, cpu:  9471630] sum_of_squares_test_ok
│ PASS [mem:  50845, cpu: 20547750] sum_of_squares_test_fail
│ PASS [after 100 tests] prop_sum_of_squares_identity
│ PASS [after 100 tests] prop_sum_of_squares_non_negative
│ PASS [after 100 tests] prop_sum_of_squares_length
└────────── with --seed=3568912749 → 7 tests | 7 passed | 0 failed

┌ merkelized_validator_example ──────────────────
│ PASS [mem:  59918, cpu: 23184608] spend_validator
│ PASS [mem:  24843, cpu: 10213348] withdraw_validator
│ PASS [after 100 tests] prop_withdraw_validator
│ PASS [after 100 tests] prop_spend_validator
└ with --seed=3568912749 → 4 tests | 4 passed | 0 failed

┌ multi_utxo_indexer_example ────────────────────
│ PASS [mem:  49782, cpu: 20035153] spend_validator
│ PASS [after 100 tests] prop_spend_validator
│ PASS [mem: 219528, cpu: 84736029] withdraw_validator
│ PASS [after 100 tests] prop_withdraw_validator
└ with --seed=3568912749 → 4 tests | 4 passed | 0 failed

┌ multi_utxo_indexer_one_to_many_example ────────
│ PASS [mem:  49782, cpu: 20035153] spend_validator
│ PASS [after 100 tests] prop_spend_validator
│ PASS [mem: 201831, cpu: 73150467] withdraw_validator
│ PASS [after 100 tests] prop_withdraw_validator
└ with --seed=3568912749 → 4 tests | 4 passed | 0 failed

┌ singular_utxo_indexer_example ─────────────────
│ PASS [mem: 169223, cpu: 67430022] spend_validator
│ PASS [after 100 tests] prop_spend_validator
└ with --seed=3568912749 → 2 tests | 2 passed | 0 failed

┌ singular_utxo_indexer_one_to_many_example ─────
│ PASS [mem: 101883, cpu: 36938474] spend_validator
│ PASS [after 100 tests] prop_spend_validator
└ with --seed=3568912749 → 2 tests | 2 passed | 0 failed

┌ stake_validator_example ───────────────────────
│ PASS [mem:  49782, cpu: 20035153] spend_validator
│ PASS [after 100 tests] prop_spend_validator
│ PASS [mem:  13078, cpu:  4800118] withdraw_validator
│ PASS [after 100 tests] prop_withdraw_validator
└ with --seed=3568912749 → 4 tests | 4 passed | 0 failed

┌ tx_level_minter_example ───────────────────────
│ PASS [mem:  94858, cpu: 36048207] spend_validator
│ PASS [after 100 tests] prop_spend_validator
│ PASS [mem:   9414, cpu:  3240885] mint_policy
│ PASS [after 100 tests] prop_mint_policy
└ with --seed=3568912749 → 4 tests | 4 passed | 0 failed

  Summary 1516 checks, 0 errors, 0 warnings
```

Provided Patterns

Stake Validator

This module offers two functions meant to be used within a multi-validator for implementing a "coupled" stake validator logic.

The primary application for this is the so-called "withdraw zero trick," which is most effective for validators that need to go over multiple inputs.

With a minimal spending logic (which is executed for each UTxO), and an arbitrary withdrawal logic (which is executed only once), a much more optimized script can be implemented.

ENDPOINTS

`spend` merely looks for the presence of a withdrawal (with arbitrary amount) from its own reward address.

`withdraw` takes a custom logic that requires 3 arguments:

1. Redeemer (arbitrary `Data`)
2. Script's validator hash (`Hash<Blake2b_224, Script>`)
3. Transaction info (`Transaction`)

UTxO Indexers

The primary purpose of this pattern is to offer a more optimized solution for a unique mapping between one input UTxO to one or many output UTxOs.

SINGULAR UTXO INDEXER

ONE-TO-ONE

By specifying the redeemer type to be a pair of integers (`(Int, Int)`), the validator can efficiently pick the input UTxO, match its output reference to make sure it's the one that's getting spent, and similarly pick the corresponding output UTxO in order to perform an arbitrary validation between the two.

The provided example validates that the two are identical, and each carries a single state token apart from Ada.

ONE-TO-MANY

Here the validator looks for a set of outputs for the given input, through a redeemer of type `(Int, List<Int>)` (output indices are required to be in ascending order to disallow duplicates). To make the abstraction as efficient as possible, the provided higher-order function takes 3 validation logics:

1. A function that validates the spending `Input` (single invocation).
2. A function that validates the input UTxO against a corresponding output UTxO. Note that this is executed for each associated output.
3. A function that validates the collective outputs. This also runs only once. The number of outputs is also available for this function (its second argument).

MULTI UTXO INDEXER

While the singular variant of this pattern is primarily meant for the spending endpoint of a contract, a multi UTxO indexer utilizes the stake validator provided by this package. And therefore the spending endpoint can be taken directly from `stake_validator` .

Subsequently, spend redeemers are irrelevant here. The redeemer of the withdrawal endpoint is expected to be a properly sorted list of pairs of indices (for the one-to-one case), or a list of one-to-many mappings of indices.

It's worth emphasizing that it is necessary for this design to be a multi-validator as the staking logic filters inputs that are coming from a script address which its validator hash is identical to its own.

The distinction between one-to-one and one-to-many variants here is very similar to the singular case, so please refer to its section above for more details.

The primary difference is that here, input indices should be provided for the *filtered* list of inputs, i.e. only script inputs, unlike the singular variant where the index applies to all the inputs of the transaction. This slight inconvenience is for preventing extra overhead on-chain.

Transaction Level Validator Minting Policy

Very similar to the stake validator, this design pattern utilizes a multi-validator comprising of a spend and a minting endpoint.

The role of the spendig input is to ensure the minting endpoint executes. It does so by looking at the mint field and making sure a non-zero amount of its asset (where its policy is the same as the multi-validator's hash, and its name is specified as a parameter) are getting minted/burnt.

The arbitrary logic is passed to the minting policy so that it can be executed a single time for a given transaction.

Validity Range Normalization

The datatype that models validity range in Cardano currently allows for values that are either meaningless, or can have more than one representations. For example, since the values are integers, the inclusive flag for each end is redundant and can be omitted in favor of a predefined convention (e.g. a value should always be considered inclusive).

In this module we present a custom datatype that essentially reduces the value domain of the original validity range to a smaller one that eliminates meaningless instances and redundancies.

The datatype is defined as following:

```
pub type NormalizedTimeRange {
  ClosedRange { lower: Int, upper: Int }
  FromNegInf  {             upper: Int }
  ToPosInf    { lower: Int             }
  Always
}
```

The exposed function of the module (`normalize_time_range`), takes a `ValidityRange` and returns this custom datatype.

Merkelized Validator

Since transaction size is limited in Cardano, some validators benefit from a solution which allows them to delegate parts of their logics. This becomes more prominent in cases where such logics can greatly benefit from optimization solutions that trade computation resources for script sizes (e.g. table lookups can take up more space so that costly computations can be averted).

This design pattern offers an interface for off-loading such logics into an external withdrawal script, so that the size of the validator itself can stay within the limits of Cardano.
Ting with

- [ ] **Ai:ken Linked List -third party developed by anastasia abs**

Introduction.

Linked list structures leverage the EUTXO model to enhancing scalability and throughput significantly. By linking multiple UTXOs together through a series of minting policies and validators, it can improve the user experience interacting with smart contract concurrently.

This project is funded by the Cardano Treasury in Catalyst Fund 10 and is aimed at enhancing the capabilities of Cardano smart contracts in handling complex data structures.

Documentation

Linked List

The Aiken Linked List is an on-chain, sorted linked list solution designed for blockchain environments, specifically utilizing NFTs (Non-Fungible Tokens) and datums. It provides a structured and efficient way to store and manipulate a list of key/value pairs on-chain.https://

Entry Structure

Each entry in the list comprises:

- NFT: A unique identifier for each entry.
- EntryDatum: A data structure containing the key/value pair, a reference to the entry's NFT, and a pointer to the next NFT in the list.
- EntryDatum Definition

- key: A unique identifier for the entry.
- value: The value associated with the key. It can be Nothing for the head entry.
- nft: The NFT representing the entry.
- next: The NFT of the next entry in the list, or Nothing for the last entry.
- Operations

Inserting an Entry

insert entry

Insertion involves:

- Inputs: Two adjacent list entries.
- Outputs:

    The first input entry, modified to point to the new entry.

The newly inserted entry, pointing to the second input entry.

The second input entry, unchanged.

Validation Rules

- Keys must maintain the order: a < b < c, where a is the lowest, b is the new key, and c is the highest.
- The pointers must be correctly updated to maintain list integrity.
- Removing an Entry

Head Apple

Banana kiwi orange peach

remove entry

To remove an entry:

- Inputs: The entry to remove and its preceding entry.
- Output: The preceding entry is modified to point to what the removed entry was pointing to.

Utilizing NFTs as Pointers

NFTs serve as robust and unique pointers within the list. Their uniqueness is ensured by a specific minting policy related to the list's head NFT.

Key Considerations

- Efficiency: As on-chain lookups are inefficient, off-chain structures are recommended for this purpose.
- Datum Hashes: Not suitable for pointers due to the complexity of updates and security concerns.
- Security: The integrity of the list is maintained through careful minting policies and entry validation.

Advanced Features

- Forwarding Minting Policy: A feature of Plutus to control NFT minting dynamically.
- List Head: Utilizes an empty head entry for validating insertions at the start of the list.
- End-of-List Insertions: Handled by ensuring the last entry points to Nothing.

Aiken Linked List implementation

The Aiken Linked List implementation provides several functions to create and manipulate Linked List. Below is a brief overview of each function:

- init: Constructs the Linked List head
- deinit: Destructs the Linked List
- insert: Inserts a node into the linked list
- remove: Removes a node from the linked list

On-Chain Association List With Constant Time Insert-Removal

Background

Credit: Neil Rutledge

The excellent article Distributed map in EUTXO model from Marcin Bugaj describes the need for an on-chain, distributed data structure which can guarantee uniqueness of entries.

If you want to validate that something doesn't exist on chain in the EUTXO model, you do not have access to anything aside from the transaction inputs, which are limited by transaction size limits. So checking that someone hasn't voted twice, for example, would be impossible without some sort of data structure that can guarantee uniqueness on insert.

A potential solution, heavily inspired by the distributed map idea, is described below that shares the following properties with the distributed map:

Verifiable uniqueness of keys on chain
Verifiable traversal on chain
However, it has a couple key advantages:

Constant time (single transaction) insert/removal with low contention
Sorted entries
Overview

The solution consists of an on-chain, sorted, linked list of key/value entries.

Each entry in the list will consist of an NFT and a datum with the key, value and pointer to the next NFT in the list (more details explained later).

Inserting an Entry

If we want to perform an insert operation that can be validated on chain to ensure no duplicate entry occurs, we can simply create a transaction with two adjacent items as inputs like so:

The transaction outputs would be:

The first entry from the inputs, now pointing to the new entry
The new entry, pointing to the second input entry
The second entry from the inputs, unchanged
To ensure uniqueness, the script will validate that the following conditions are true:

a < b < c
input a points to input c
output a points to output b
output b points to output c
Where a = lowest input key, b = new key and c = highest input key.

As you can see, it is extremely simple to prove/validate whether or not a given key exists by inspecting two adjacent keys and checking if the new key fits between those according to the ordering.

An empty head entry can be used for validating inserts at the start of the list (i.e., the head and first entry must be inputs and the new entry must have a key lower than the first entry's key).

For inserting at the end of the list, the script can simply validate that the input entry points to nothing and that the new entry key is higher than the input entry's key.

Removing an Entry

If we need to remove an entry from the list, the process is even simpler.

The transaction inputs in this case would be the entry to be removed and the previous entry.

The removed entry NFT could be burned and the output would be the previous entry now pointing to what the removed entry was pointing to.

Wait, looking up entries would be horribly inefficient!

Yes, it is true that a linked list is a very inefficient data structure for performing lookups. But since lookups will be done off-chain, it doesn't really matter. A more efficient data structure could be maintained off-chain for lookups if that's required. The key here is that the operations on chain only take a single transaction and as few inputs/outputs as possible.

Datums and Redeemers

The following are examples of what the datum and redeemer types could look like.

```
type NFT = AssetClass

data EntryDatum = EntryDatum
{ key   :: BuiltinByteString
, value :: Maybe SomeValue
, nft   :: NFT
, next  :: Maybe NFT
}

data Redeemer
= Insert SomeValue
| Use
| Update SomeValue
| Remove
```

The EntryDatum holds the key/value pair for each entry. In the case of the head entry, the value would be Nothing. Each EntryDatum also points to the NFT that identifies the entry, as well as the next NFT in the list that this entry points to (which is Nothing for the last entry).

Each EntryDatum could also hold the key of the next entry, thus eliminating the need to pass in the second entry when performing an insert (since only the key is required for validation and nothing changes for that entry).

NFTs as Pointers

An important question now is: where do the NFTs come from and how can we ensure they are unique?

One way to achieve this would be to first mint the head NFT and then use its AssetClass or CurrencySymbol as a parameter for both the entry NFT minting policy and the script that locks them. This will create a unique script hash for each list as well as unique NFT CurrencySymbols for each list. The minting policy can validate that either the head NFT is spent or an entry that it minted is spent before minting any further entry NFTs. The script can also validate that each NFT added has the correct AssetClass (they will all share the same CurrencySymbol but should have a unique TokenName such as a hash of the key).

Plutus also has a forwardingMintingPolicy (described here: Forwarding Minting Policy) that could potentially work.

How About Using Datum Hashes as Pointers?

Using the the DatumHash sounds very convenient as a pointer, but there is a problem in that updating one entry will change its hash, thus requiring the parent's pointer to be updated, as well as its parent, and so on all the way up to the head entry.

It may then be tempting to use a hash of all fields, excluding the next field, but this has another problem: someone could easily create a malicious EntryDatum that hashes the same as another but points to some sublist of malicious entries.

So it seems using datum hashes is out of the question as a standalone option (though this could work in combination with NFTs).

Folding/Traversing Over the List

Since the data structure forms a linked list, folding/traversing is extremely straight forward. However, if you need to verify that the entire list has been traversed on chain, you'll need some way of building up a proof from several transactions.

Let's look at how we would prove that the entire list has been folded over using a new datum.

```
data FoldingDatum a = FoldingDatum
{ start :: NFT
, next  :: NFT
, nft   :: NFT
, accum :: a
}
```

To create this FoldingDatum, you would spend one or more list entry UTXOs in a transaction. A validation rule would check that the spent entry UTXOs form a valid linked list and that the output FoldingDatum contains:

start = the first input entry's nft field
next = the last input entry's next field
nft = a reference to a new NFT identifying the FoldingDatum and providing proof that it is valid
accum = whatever value you want to accumulate
The FoldingDatum can now be used in another transaction that starts with the next entry and consumes some number of additional entries (however many fit in the transaction). The start value will remain the same throughout this process, but the next will point further down the list after each transaction until reaching Nothing at the end of the list.

Parallel Folds

Folding over a list using a single datum over a series of blocks is certainly an option, but we can do better. Instead of creating a single FoldingDatum, we can create n FoldingDatums in parallel that each consume different sections of the list.

Imagine that there is a list of 1000 entries and 100 FoldingDatums are created that each consume 10 entries as 100 parallel transaction. The following example will represent the FoldingDatums as (start, next) tuples.

(head, 10) (10, 20) (20, 30) (30, 40) (40, 50) (50, 60) ... (990, Nothing)
In another block, assuming we can fit 10 inputs per transaction, the 100 FoldingDatums could themselves be folded into 10 FoldingDatums.

(head, 100) (100, 200) (200, 300) (300, 400) (400, 500) ... (900, Nothing)
These 10 FoldingDatums can finally be folded into a single datum.

(head, Nothing)

Since this whole process was controlled by validation rules that confirm the validity of each FoldingDatum, the final result serves as a proof that all entries were included, which can be used by a smart contract to verify on chain that all votes have been counted, for example. And this whole process happened in O(log n) time:

Constant time creation of FoldingDatums (parallel transactions in one block)

Logarithmic time merging of FoldingDatums

The off-chain logic will have to traverse the list in O(n) time in order to create the transactions involved but, again, we aren't really concerned with the off-chain efficiency.

Note: For really big lists, you'll be bumping up against the fundamental TPS limits of the blockchain, so some of these operations will take additional blocks.

Handling Contention / Race Conditions

One additional thing that needs to be considered here is how to handle contention on UTXOs during the folding/traversal process. This could result in failed transactions as well as unexpected behaviour from the list changing part way through.

A few potential options:

Require that the head token is spent during each insert/update/removal and put a lock field in the datum to allow locking the list in a read-only mode. (Need to consider when the list is allowed to be unlocked, whether there should be a timeout, etc.)

Similar to the first option, require the head token to be spent for each insert/update/removal, but instead of using a lock field, maintain an incrementing version number in the head datum as well as the version of each entry. This way, a version number can be declared in the FoldingDatum, which would allow skipping over entries with higher version numbers. Deleted entries would need to be kept around with some sort of deleted status.

(h v5) -> (apple v3) -> (banana v1) -> (kiwi v2 deletedInV4) -> (kiwi v5)

Don't lock the list or require spending of the head entry (which will have contention on it making it difficult to even set the lock) and instead include approximate timestamps on each entry. With a few modifications (such as including the ability to have entries branch off to different versions) this could be turned into an immutable data structure that maintains the full version history.

Note: If there is a need for shared state that is limiting throughput, things like queuing and batching can be introduced as well.

Summary

This document glossed over some details for the sake of brevity but it has outlined what I feel is an elegant approach to working with on chain data that has uniqueness guarantees and is also extremely efficient to maintain on-chain.

The ' Results Testing

```
┌── linkedlist/tests ────────────────────────────────────────────
│ PASS [mem:  3005, cpu:  2072442] div_ceil_1
│ PASS [mem:  3407, cpu:  2370919] div_ceil_2
│ PASS [mem:  3005, cpu:  2072442] div_ceil_3
│ PASS [mem:  3005, cpu:  2072442] div_ceil_4
└──────────────────────────────── 4 tests | 4 passed | 0 failed
```

```
┌─ sample ──────────────────────────────────────────────
│ PASS [mem: 361900, cpu: 135081986] mint_validator_init
│ PASS [mem: 230823, cpu:  84464591] mint_validator_deinit
│ PASS [mem: 210870, cpu:  79532791] mint_validator_deinit_fails_on_non_empty
│ · with traces
* | expect Empty = head_node.node.next
│ PASS [mem: 690695, cpu: 275294314] mint_validator_insert
│ PASS [mem: 648665, cpu: 260721199] mint_validator_remove
└──────────────────────────────────────────────── 5 tests | 5 passed | 0 failed
```

Functionalities and features of aiken lang development

Basic functions

1. State Management:

- Data Storage: Store and retrieve state information relevant to the contract.
- State Transitions: Manage changes to state based on contract interactions.

2. Token Management:

- Minting & Burning: Create and destroy tokens, including ADA or custom tokens (native tokens).
- Transfers: Facilitate the transfer of tokens between parties.
- Authentication & Authorization:

Owner Checks: Verify the identity of parties interacting with the contract.
Access Control: Restrict certain functions to specific users or roles.
Condition-Based Execution:

If-Else Logic: Execute code based on conditional logic.

Thresholds & Limits: Implement limits or thresholds for transactions or state changes.
Advanced Features
Multi-Signature Requirements:

Multi-Sig Transactions: Require multiple parties to sign off on a transaction or state change.
Time-Based Constraints:

Timelocks: Specify conditions under which actions can be executed based on time.
Expiration Dates: Set deadlines for contract actions or states.
Decentralized Governance:

Voting Mechanisms: Implement voting systems for decision-making processes within the contract.
Proposal Handling: Allow for the creation and handling of proposals or changes to contract terms.
Automated Payments:

Scheduled Payments: Automate recurring payments or transfers.
Conditional Payments: Trigger payments based on predefined conditions or milestones.
Oracles Integration:

External Data Feeds: Fetch and use data from external sources (e.g., price feeds) to influence contract behavior.
Interoperability:

Cross-Chain Interactions: Facilitate interactions between Cardano and other blockchains or systems.
Security Features
Formal Verification:

Proofs: Ensure the contractâs correctness through mathematical proofs or formal methods.
Error Handling:

Reverts & Rollbacks: Implement mechanisms to revert changes in case of errors or failed conditions.
Gas Management:

Cost Control: Optimize the contract to minimize transaction fees and resource consumption.
Audit Trails:

Logs & Records: Maintain detailed logs of all interactions and state changes for audit purposes.
User Experience Enhancements
User-Friendly Interfaces:

Front-End Integration: Develop user interfaces for easier interaction with the smart contract.
Customizable Parameters:

Adjustable Settings: Allow users to adjust certain parameters or settings within the contract.
Notifications:

Alerts & Updates: Notify users of important events or changes related to the contract.
Development Considerations
Modular Design:

Reusable Components: Design contracts in a modular way to facilitate reuse and maintenance.
Upgradability:

Upgradeable Contracts: Implement mechanisms to upgrade or patch contracts without losing existing state.
Testing & Simulation:

Testnets: Utilize Cardanoâs testnets for thorough testing before deployment.
Simulation Tools: Use simulation tools to model and predict contract behavior under various conditions.