

# CardanoPyC

**Project: Open-Source Review and Reputation System - with a Use Case for Hotels**

**Project Catalyst Fund 13**

**Milestone 1: The Review System: UI Design and Tech Design**

## Contents

<b>1. Introduction</b>	3
<b>2. Scope</b>	3-6
2.1. Syntax highlighting and code completion	
2.2. Cardano Blockchain Explorer API Integration	
2.3. Smart Contract Deployment/Connecting Node	
2.4. Wallet Management	
2.5. Debugging Tools	
<b>3. Technical Specifications for CardanoPyC Extension.</b>	7-21
3.1. Setup Environment:	
3.2. Develop Syntax Highlighting and Code Completion	
3.3. Implement Cardano Blockchain Explorer API Integration	
3.4. Smart Contract Deployment from Pycharm IDE	
3.5. Wallet Management	
3.6. Debugging Tools for Smart Contracts and Wallet Transactions.	
<b>4.UI/UX Mockups</b>	22-26
<b>5. Architecture diagram</b>	27
<b>6. Conclusion</b>	28

## 1.Introduction

**CardanoPyC** is a PyCharm Integrated Development Environment (IDE) extension designed to enhance the development experience for **Plutus** and **Haskell** programmers by offering the following key features:

- Syntax Highlighting & Code Completion.
- Cardano Blockchain Explorer Integration.
- Smart Contract Deployment
- Wallet Management
- Cardano node connection
- Debugger tools

## 2.Scope

### 2.1. Syntax highlighting and code completion:

The extension will provide advanced syntax highlighting and intelligent code completion tailored for the Plutus programming language, aiming to enhance developer productivity and streamline the coding experience in PyCharm.

#### Features:

##### I. Syntax Highlighting:

- Offers visually distinct color-coding for functions, operators, variables, keywords, and other elements.
- Ensures a clear, visually appealing representation, consistent with PyCharm's syntax highlighting for other languages.
- Allow users to customize highlighting colors and styles through the **IDE's Settings > Editor > Colour Scheme menu.**

The extension will provide visually distinct and color-coded highlighting for the following elements:

**Keywords:** Reserved words in the Plutus language, color roman coffee.

**Data Types:** Built-in and custom-defined data types, color plum purple.

**String:** Basic string in the Plutus, color grey asparagus.

**Comments:** Inline and block comments, color grey.

**Pragma:** Inline Pragas, color light yellow.

## II. Code Completion:

- Provides context-aware suggestions to assist users in writing accurate and efficient code.

The extension will offer intelligent, context-aware suggestions, including:

**Variable Names:** Recommend existing variable names based on scope and usage.

**Data Types:** Suggest relevant built-in or user-defined data types.

**Code Snippets:** Predefined templates for common coding patterns to save time and improve efficiency.

## 2.2. Cardano Blockchain Explorer API Integration

- **Objective:** Query and display blockchain data directly within the PyCharm extension.

- **Steps:**

1. **Choose an Explorer:** Use **CardanoScan**, one of the most popular and widely used blockchain explorers.
2. **Connect API:** Integrate with the **CardanoScan API** to fetch data such as transactions, addresses, and block details.
3. **Display Data:** Present the fetched data in an intuitive format within the PyCharm terminal, ensuring easy accessibility for developers.

## 2.3. Smart Contract Deployment / Node Connection

**Objective:** Simplify the process of deploying Plutus smart contracts and connecting to the Cardano blockchain from within PyCharm.

### Steps:

#### Smart Contract Deployment:

- Build a feature to allow users to deploy Plutus smart contracts directly from PyCharm.
- Integrate deployment commands to work seamlessly with the Cardano CLI.

#### Node Connection:

- Enable developers to connect to a **Cardano node** within the PyCharm environment.
- Provide configuration settings for connecting to a public or private node, based on user preference.

## 2.4. Wallet Management

- **Objective:** Enable users to manage Cardano wallets directly within the extension.
- **Features:**
  - **Wallet Creation:** Provide an option to create new wallets within the extension.
  - Provide functionality for checking wallet balances, sending transactions, and using the wallet.
  - **View Balances and Transactions:** Display wallet balances and transaction history in an organized manner.
- **Implementation:**
  - Integrate with Cardano wallet APIs or directly interact with the blockchain to retrieve wallet details.

## 2.5. Debugging Tools

- **Objective:** Facilitate debugging for Plutus smart contracts and transaction processing.
- **Steps:**
  1. **Step-through Debugging:** Allows developers to execute smart contracts.
  2. **Transaction Debugging:** Include features to analyze transaction inputs, outputs, and execution results.
  3. **Error Insights:** Display detailed error messages.

## 3. Technical Specifications for CardanoPyC Extension

### 3.1. Setup Environment

#### 1. Scaffold the Project:

- Use the **IntelliJ Plugin Development Kit (IntelliJ platform plugin sdk)** to create a boilerplate project structure for the extension.

Run the following commands to initialize the project:

```
./gradle buildPlugin
```

- This creates the necessary project structure and configuration files for a PyCharm plugin.

#### 2. Review the Project Structure:

- Ensure the required files (`plugin.xml`, `build.gradle.kts`) are set up correctly for development.

### 3.2. Develop Syntax Highlighting and Code Completion

#### 1. Lexer Generation.

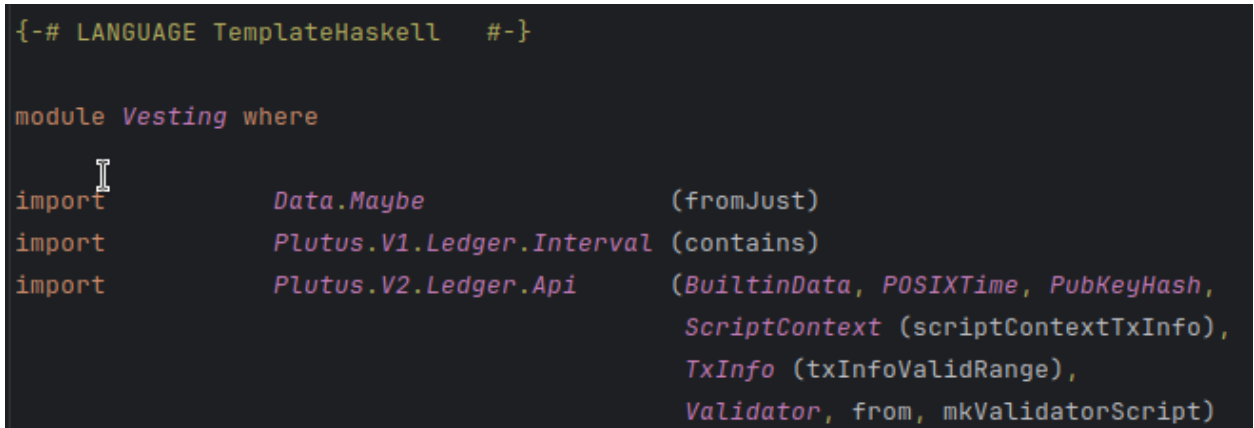
- Create a lexer to tokenize the code into meaningful elements.
- Use a `.flex` file to define patterns for Haskell/Plutus language tokens (e.g., keywords, identifiers, and operators).
- Compile the lexer using IntelliJ's **JFlex** tool.

#### 2. Token Mapping:

- Map the lexer tokens to specific **TextAttributes** using IntelliJ's syntax highlighting API.

### 3. Highlighting Integration

- Implement a syntax highlighter that maps PSI elements (Program Structure Interface) to corresponding token types.
- Ensure the highlight supports customization via Pycharm's color scheme settings.



```
{-# LANGUAGE TemplateHaskell #-}

module Vesting where

import Data.Maybe (fromJust)
import Plutus.V1.Ledger.Interval (contains)
import Plutus.V2.Ledger.Api (BuiltinData, POSIXTime, PubKeyHash,
                             ScriptContext (scriptContextTxInfo),
                             TxInfo (txInfoValidRange),
                             Validator, from, mkValidatorScript)
```

Syntax Highlight UI Mockup

### 4. Code Completion:

#### Basic Completion

- Suggest identifiers, keywords, and functions relevant to the current cursor position.
- Include Plutus-specific elements such as **txSignedBy** and **ScriptContext**.

#### Implementation Details

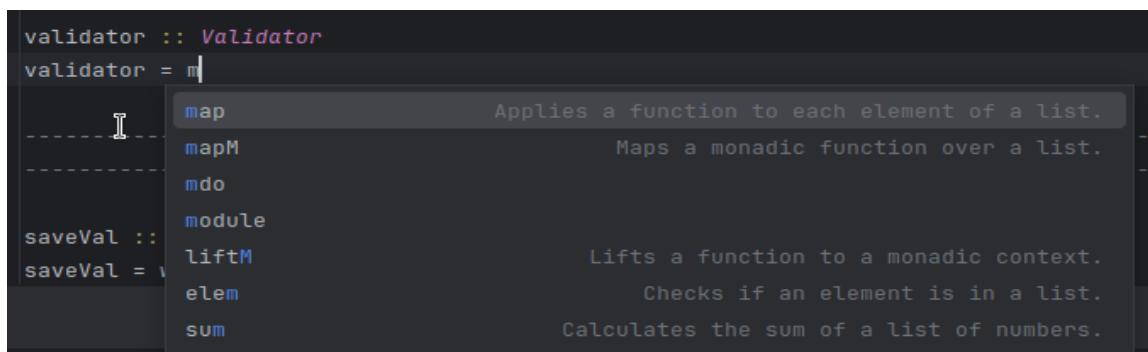
##### ● Completion Workflow

- Use IntelliJ's **CompletionContributor API** to hook into the completion system.
- Analyze the context of the code using the PSI tree to determine applicable suggestions.



- **Data Sources for Suggestions**

- Predefined keywords and operators.
- Available types, functions, and variables in the current scope.
- Standard Haskell libraries and Plutus modules.



Code Completion UI Mockups

### 3.3. Implement Cardano Blockchain Explorer API Integration

To enable efficient Cardano blockchain exploration within the PyCharm IDE, the following steps and APIs will be utilized:

- **Objective:**

Integrate Cardano Blockchain Explorer API functionality into PyCharm to retrieve essential blockchain data. This integration will allow developers to interact with the Cardano blockchain seamlessly from within the IDE.

- **API Key Requirement:**

A free API key from CardanoScan is required to access their services. Each user must provide their own API key, which can be obtained by signing up on the CardanoScan website.

- **Implementation Plan in PyCharm:**

1. Create a configuration panel within PyCharm to allow users to input and store their CardanoScan API key securely.
2. Develop backend API integration within PyCharm plugins or tools to call the CardanoScan endpoints.

3. Display retrieved data in a structured and developer-friendly manner using PyCharm's custom terminal.

### **Free APIs:-**

#### **Get Block Details:**

Query Parameter: Block hash(string), blockHeight(integer), absoluteSlot(integer), epoch(integer), slot(integer).

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/block
```

Bash Example: Get block details EndPoint

#### **Get the latest block details:**

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/block/latest
```

Bash Example: Get the latest block details endpoint

#### **Get address balance:**

Query parameters: Address(integer).

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/address/balance?address=string
```

Bash Example: Get address balance endpoint

#### **Get pool details:**

Query parameters: PoolID(string)

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/pool?poolId=string
```

Bash Example: Get pool details endpoint

### **Get pool stats:**

Query parameters: poolId

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/pool/stats?poolId=string
```

Bash Example: Get pool stats endpoint

### **Get pools list:**

Query parameters: pageNo(integer), search(string), retiredPools(boolean), sortBy(string), order(string), limit (integer).

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/pool/list?pageNo=42
```

Bash Example: Get pools list endpoint

### **Get pools that are set to expire:**

Query parameters: pageNo(integer), limit(integer).

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/pool/list/expiring?pageNo=42
```

Bash Example: Get pools that are set to expire

### Get expired pools:

Query parameters: pageNo(integer), limit(integer).

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/pool/list/expired?pageNo=42
```

Bash Example: Get expired pools endpoint

### Get asset details:

Query parameters: assetId(string), fingerprint(string).

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/asset
```

Bash Example: Get asset details endpoint

### Get transaction details:

Query parameters: hash(string). Get transaction details: Query parameters: hash(string).

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/transaction?hash=string
```

Bash Example: Get transaction details endpoint

**Get stake key details:**

Query parameters: rewardAddress(string).

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/rewardAccount?rewardAddress=string
```

Bash Example: Get stake key details endpoint

**Get network details:**

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/network/state
```

Bash Example: Get network details endpoint

**Get network protocol details:**

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/network/protocolParams
```

Bash Example: Get Network protocol details endpoint

**Get CCHot details:**

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/governance/ccHot?hotHex=string
```

Bash Example: Get CCHot details endpoint

#### Get CCMember details:

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/governance/ccMember?coldHex=string
```

Bash Example: Get CCMember details endpoint

#### Get Committee Information:

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/governance/committee
```

Bash Example: Get Committee Information endpoint

#### Get Governance Action:

Query parameters: actionId(string).

```
$ curl \
-X GET https://api.cardanoscan.io/api/v1/governance/action?actionId=string
```

Bash Example: Get Governance Action endpoint

### 3.4. Smart Contract Deployment in PyCharm IDE

#### Connecting to a Node

**Objective:** Enable users to execute Cardano CLI commands directly within PyCharm for seamless blockchain interaction.

**Steps:****1. Node Configuration Setup:**

- Create a dedicated settings panel in PyCharm where users can configure their local Cardano node environment.

**2. Display Responses:**

- Fetch the results of the executed CLI command and present them to the user as a PyCharm notification or in a dedicated output panel.

**Smart Contract Deployment**

- Allow users to execute Cardano CLI directly from within Pycharm. Commands such as Transaction list by address, stake key details, etc.
- Prompt user input or use a for the username, password, IP address, and port number.
- Finally, display the response as a notification.

**Steps:****1. Select a .plutus File:**

- Provide users with an option to select a Plutus file.

**2. Deploy Smart Contracts:**

- If the user's environment is connected to a Cardano node, enable deployment of the smart contract directly to the blockchain.
- Use the Cardano CLI with addresses and keys stored in PyCharm's **local storage state** or configuration files.

**3. Error Handling:**

- Provide detailed error messages if the build or deployment fails due to issues like Cabal configuration errors or missing dependencies.
- Suggest potential fixes or guide the user in correcting their setup.

### 3.5. Wallet Management

**Objective:** Provide an intuitive interface within PyCharm for managing Cardano wallets, enabling users to interact with different networks, create wallets, restore wallets, and perform essential wallet operations like sending, receiving, and checking balances.

#### Steps

##### 1. Accessing Wallet Management Tools:

- Add a **dedicated sidebar button** or **menu command** in PyCharm for wallet management.
- Clicking this option opens a **Wallet Management UI** integrated within the PyCharm interface.

##### 2. On-Chain Connection:

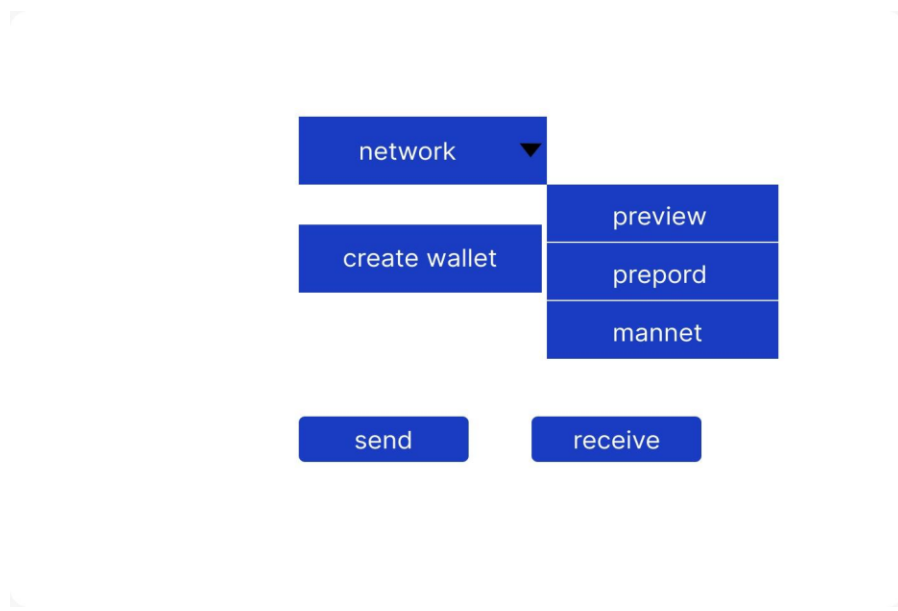
- Integrate on-chain connection options using **Lucid Cardano**, **Blockfrost API**, or **Cardano-CLI**.
- Provide step-by-step documentation for both command-line (CLI) usage and the GUI interface to ensure users understand how to configure and connect their wallets to the blockchain

#### Network Selection

##### 3. Network Switching:

- Allow users to switch between networks (e.g., **Preview**, **Preprod**, and **Mainnet**) using the Wallet Management UI.
- For GUI users: Enable network selection directly via dropdown menus powered by **Lucid Cardano**.
- For CLI users: Document the steps for changing the node configuration file to match the desired network.





Wallet Network Switching UI Mockups

## Wallet Creation

### 4. Seed Phrase Generation:

- Add functionality for users to create new wallets.
- Display a **12/24-word seed phrase** for secure storage.
- Educate users with on-screen tips or tooltips on securely storing their seed phrases.

## Wallet Restoration

### 5. Restoring Wallets:

- After selecting a network, allow users to restore wallets by entering their **seed phrase** via the Wallet Management UI.

- Validate the entered seed phrase before proceeding to restore the wallet.

## Core Wallet Functionalities

### Perform Wallet Operations:

- Enable core wallet functionalities directly within the PyCharm IDE, including:
- **Sending ADA or tokens.**
- **Receiving funds** (displaying a wallet address for a QR code or copy).
- **Checking balances** and transaction history in a tabular or graphical format.



Wallet Core Functionalities UI Mockup

## 3.6. Debugging Tools for Smart Contracts and Wallet Transactions

### Specifications

#### Plutus Smart Contract Debugging

- **Step-through Debugging**

- **Integration with GHCi and Haskell Debugger:**

Utilize GHCi and the Haskell debugger to allow step-by-step execution of Plutus smart contracts directly within the devcontainer.

- **Workflow in PyCharm:**

Users can open the Plutus development environment by launching the Devcontainer Open Feature in PyCharm.

Clicking the Debug button in the CardanoPyC plugin starts the debugging process.

- **Interactive Debugging**

- **Local Debugging in GHCi:**

Enable users to interactively debug Plutus smart contracts in GHCi within the devcontainer.

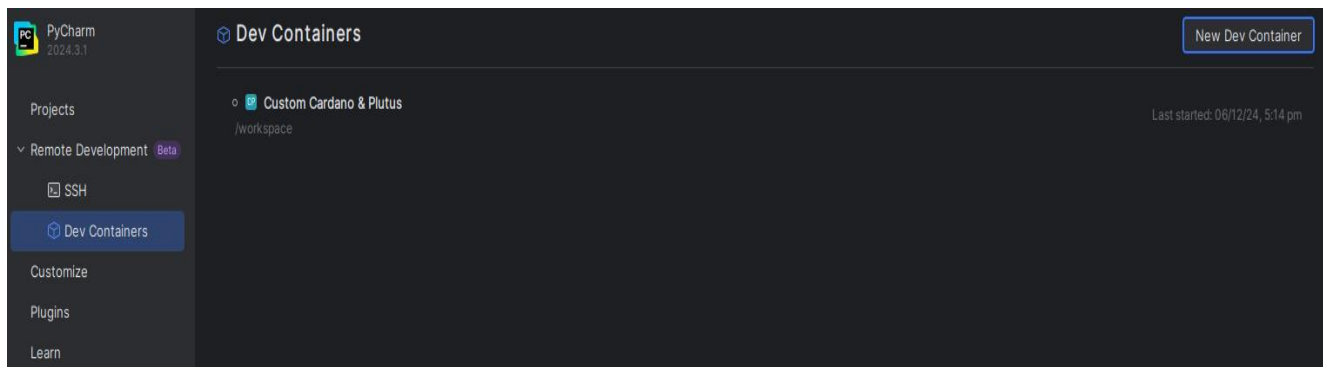
- **Real-Time Feedback:**

Errors, warnings, and evaluation results will appear in real time in PyCharm's debug console.

## User Workflow

### 1. Open Devcontainer in PyCharm:

- The user opens the Plutus development environment using the Devcontainer Open Feature in PyCharm professional, which automatically sets up the required tools and libraries.



Open Devcontainer UI Mockup

### 2. Load and Debug Code

- Within the devcontainer, the user loads the Plutus smart contract code.
- Clicking the Debug button in the CardanoPyC plugin starts the debugging session using the GHCi interpreter.

```
Build profile: -w ghc-8.10.7 -01
In order, the following will be built (use -v for more details):
- week02-0.1.0.0 (lib) (file lecture/Burn.hs changed)
Preprocessing library for week02-0.1.0.0..
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
[7 of 7] Compiling Burn          ( lecture/Burn.hs, /workspace/code/dist-newstyle/build/x86_64-linux/ghc-8.10.7/week02-0.1.0.0/build/Burn.o )
Ok, 7 modules loaded.
Prelude Gift>
```

Lode Code UI Mockup

### 3. Real-Time Debugging:

- Step through the contract code and interact with it in GHCi.
- Debug information, such as evaluation results, errors, and warnings, will be displayed dynamically in PyCharm's debug console.

### 4. Inspect and Refine

- The user inspects the code, makes necessary modifications, and re-runs the debugger as needed.

## Deliverables

### ● Devcontainer Integration:

- A pre-configured devcontainer for Plutus development accessible via PyCharm's Devcontainer Open Feature.

### ● Debugging Tools:

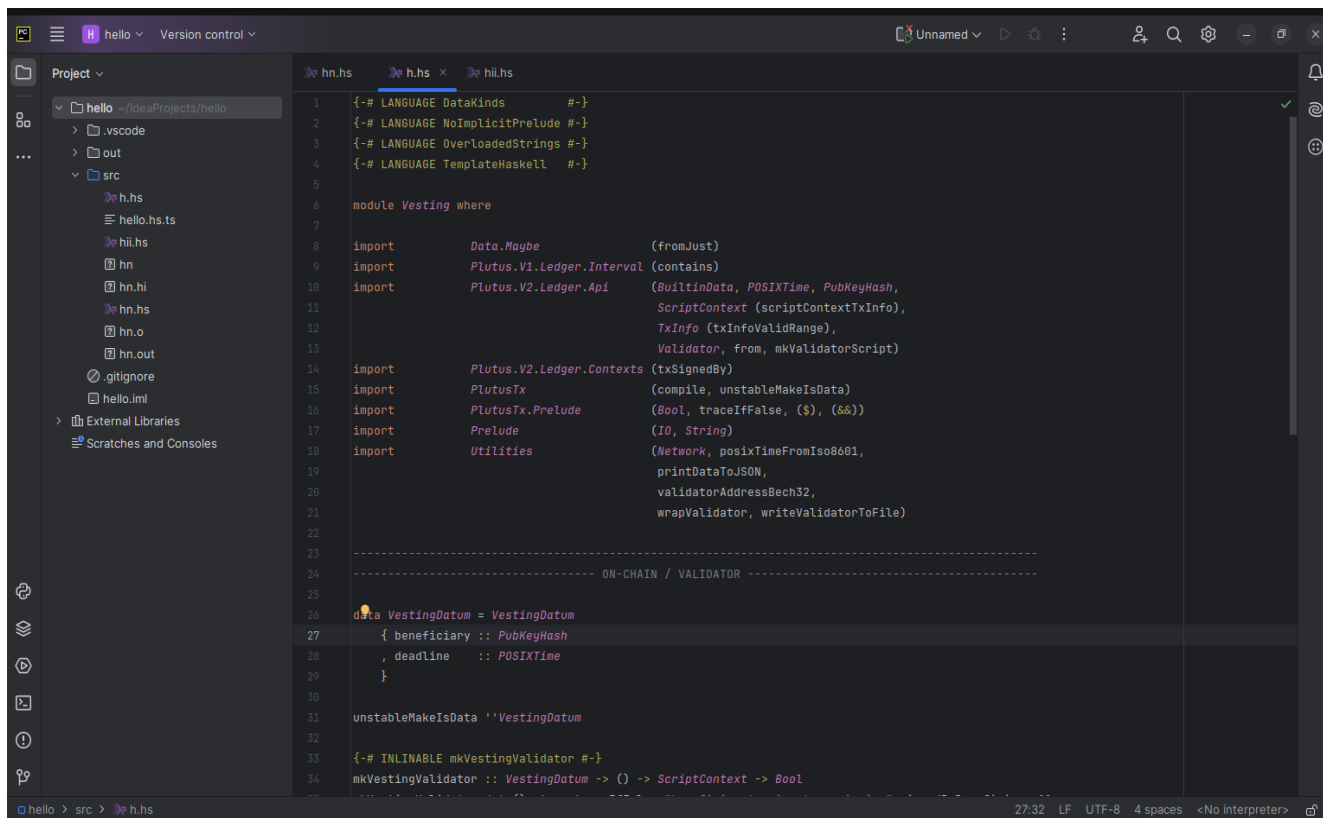
- Full integration with GHCi for step-through and interactive debugging.

### ● Enhanced Debug Console:

- Real-time display of evaluation results, warnings, and errors within PyCharm.

## 4. UI/UX Mockups.

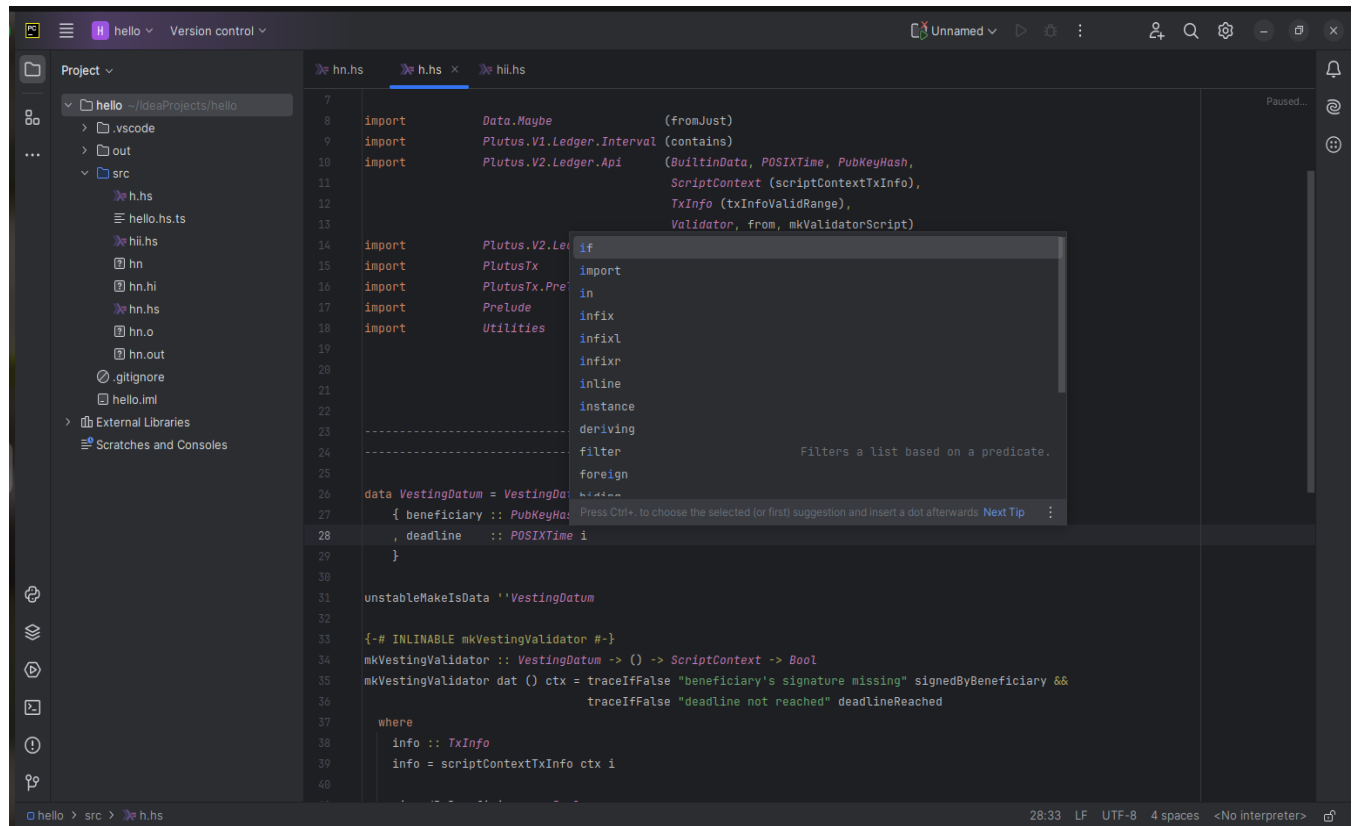
### Syntax Highlighting:



The screenshot shows an IDE window with a project named 'hello' and a file 'h.hs' open. The code is written in a Haskell-like syntax and is fully syntax-highlighted. The code defines a module 'Vesting' and includes several imports from the 'Plutus' library. It defines a data type 'VestingDatum' and a function 'mkVestingValidator'.

```
1 {-# LANGUAGE DataKinds #-}
2 {-# LANGUAGE NoImplicitPrelude #-}
3 {-# LANGUAGE OverloadedStrings #-}
4 {-# LANGUAGE TemplateHaskell #-}
5
6 module Vesting where
7
8 import Data.Maybe (fromJust)
9 import Plutus.V1.Ledger.Interval (contains)
10 import Plutus.V2.Ledger.Api (BuiltinData, POSIXTime, PubKeyHash,
11                               ScriptContext (scriptContextTxInfo),
12                               TxInfo (txInfoValidRange),
13                               Validator, from, mkValidatorScript)
14 import Plutus.V2.Ledger.Contexts (txSignedBy)
15 import PlutusTx (compile, unstableMakeIsData)
16 import PlutusTx.Prelude (Bool, traceIfFalse, ($), (&))
17 import Prelude (IO, String)
18 import Utilities (Network, posixTimeFromIso8601,
19                  printDataToJSON,
20                  validatorAddressBech32,
21                  wrapValidator, writeValidatorToFile)
22
23 ----- ON-CHAIN / VALIDATOR -----
24
25 data VestingDatum = VestingDatum
26   { beneficiary :: PubKeyHash
27   , deadline   :: POSIXTime
28   }
29
30 unstableMakeIsData ''VestingDatum
31
32 {-# INLINABLE mkVestingValidator #-}
33 mkVestingValidator :: VestingDatum -> () -> ScriptContext -> Bool
```

## Code Completion:



## CardanoScan Api Data:-

The screenshot shows an IDE with a project named 'hello' and a file 'h.hs' open. A context menu is open over the 'CardanoApi' module, showing options like 'Fetch Latest Block', 'Tasks & Contexts', 'View PSI Structure...', 'Code With Me...', 'Create Command-line Launcher...', 'Create Desktop Entry...', 'Services', 'Internal Actions', 'XML Actions', 'Markdown', 'Python or Debug Console', 'Sync Python Requirements...', 'Create setup.py', 'Run setup.py Task...', and 'Qodana'.

```

28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61

```

The screenshot shows the same IDE with the 'h.hs' file open. A context menu is open over the 'CardanoApi' module. Below the code editor, a terminal window displays Cardano block data.

```

22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

```

```

{
  "hash": "b1a323eab760542e4ce8f4ad6b7b6855e5ae4de3ed8ca084145258c22b0b9d2",
  "previousBlockHash": "dd585ca1ad7c4982613f25d426b129601f2c41cc0314a39755cd46492d01ac28",
  "blockHeight": 1.1226986E7,
  "totalFees": "0",
  "slot": 52595.0,
  "epoch": 528.0,
  "absSlot": 1.42785395E8,
  "timestamp": "2024-12-16T12:21:26.000Z",
  "txCount": 0.0,
  "assetTxCount": 0.0,
  "totalOutput": "0",
}

```



## Deployment:

The screenshot shows a code editor with a dark theme. The left sidebar displays a project structure with folders 'cogs' and 'bot.plutus', and a file '.env'. The main editor area has two tabs: 'testing.hs' and 'test.hs'. The 'testing.hs' tab is active, showing a Haskell smart contract for a vesting contract. The code includes imports for 'Data.Maybe', 'Plutus.V2.Ledger.Interval', and 'Plutus.V2.Ledger.Api', and defines a 'VestingDatum' type with fields for 'beneficiary' and 'deadline'.

```

1 {-# LANGUAGE DataKinds    #-}
2 module Vesting where
3 import Data.Maybe (fromJust)
4 import Plutus.V2.Ledger.Interval (contains)
5 import Plutus.V2.Ledger.Api      (BuiltInData, POSIXTime, PubKeyHash)
6
7 ----- ON-CHAIN / VALIDATOR -----
8
9 data VestingDatum = VestingDatum
10 { beneficiary :: PubKeyHash
11   , deadline   :: POSIXTime
12 }

```

Below the code editor is a terminal window showing the deployment process. The logs indicate that the smart contract was successfully validated, compiled to bytecode, and deployed to the Cardano blockchain.

```

>>> Initializing CardanoPy Smart Contract Deployment...
>>> Environment: Local Devcontainer
>>> Node Connection: Connected (localhost:3001)
>>> Wallet: UserWallet (Balance: 100 ADA)
>>> Smart Contract: sample-contract.plutus
>>> Transaction Type: Smart Contract Deployment

[INFO] Step 1/4: Validating Smart Contract...
[PASS] Validation Successful: No syntax or compilation errors detected.

[INFO] Step 2/4: Compiling Smart Contract...
[PASS] Compilation Successful: sample-contract.plutus compiled to bytecode.
      Output: ./build/sample-contract.plc

[INFO] Step 3/4: Preparing Deployment Transaction...
      - Smart Contract Address: addr1qxyz...abc
      - Transaction Fee: 0.2 ADA
      - Collateral UTXOs: 2 selected
[PASS] Transaction Prepared Successfully.

[INFO] Step 4/4: Signing and Submitting Transaction...
      - Using Wallet: UserWallet
      - Signing Keys Applied.
[PASS] Transaction Submitted Successfully!
      - Transaction ID: d4b7a5f...f8e9

>>> Deployment Complete!
>>> Smart Contract Address: addr1qxyz...abc
>>> Monitor the status using the Cardano Blockchain Explorer.

```

## Debugging:

The screenshot shows the Plutus IDE interface. On the left, the Project Explorer shows a project named 'cogs' with a file 'bot.plutus'. The main editor displays a Haskell script 'testing.hs' with the following content:

```

1 {-# LANGUAGE DataKinds     #-}
2 module Vesting where
3 import Data.Maybe (fromJust)
4 import Plutus.V2.Ledger.Interval (contains)
5 import Plutus.V2.Ledger.Api      (BuiltinData, POSIXTime, PubKeyHash)
6
7 ----- ON-CHAIN / VALIDATOR -----
8 data VestingDatum = VestingDatum
9
10 { beneficiary :: PubKeyHash
11   , deadline   :: POSIXTime
12 }

```

Below the script, the Debug Output pane shows the following log messages:

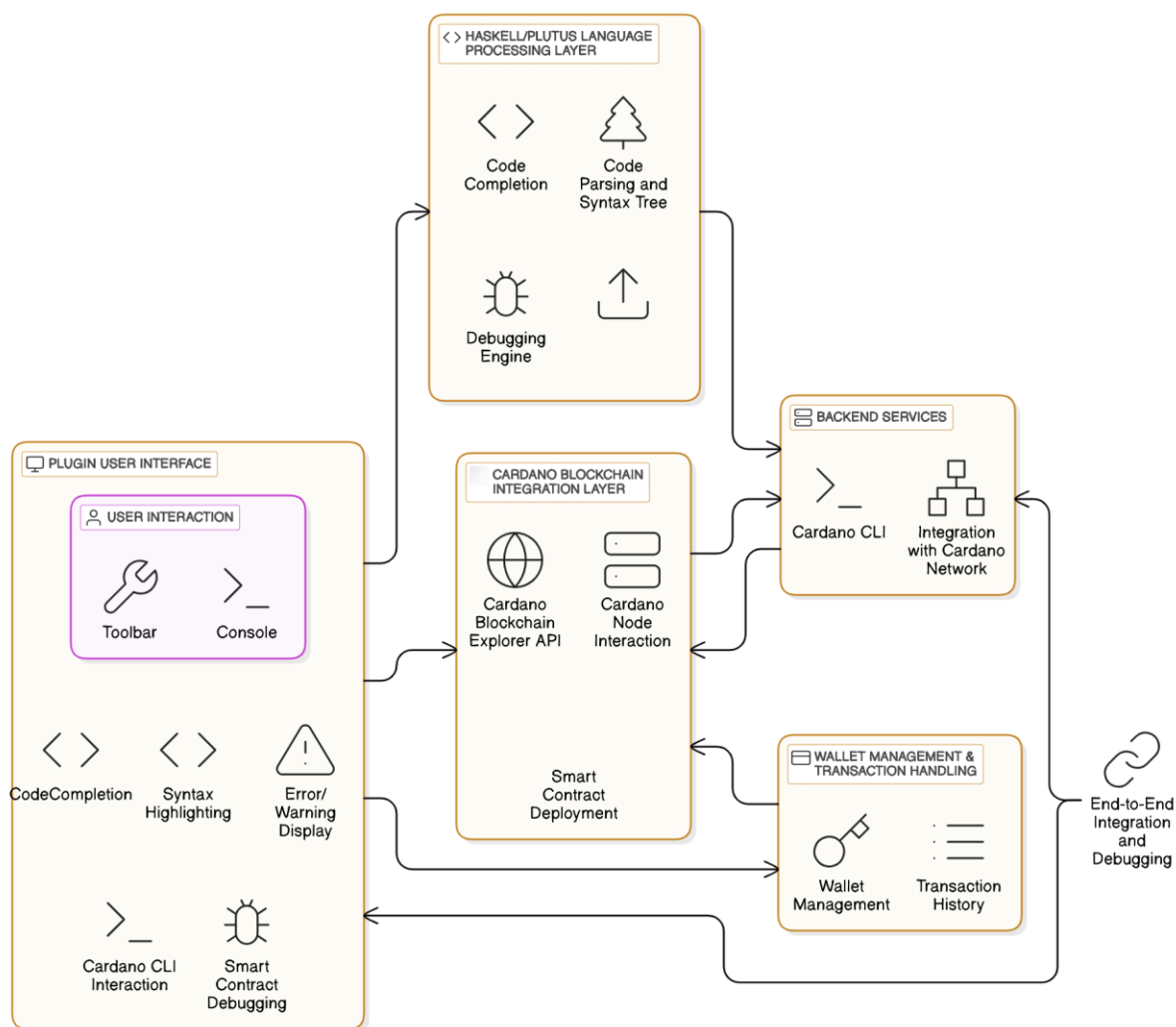
```

[INFO] Debugger initialized.
[INFO] Connecting to Plutus Debug Adapter...
[DEBUG] Debug Adapter Protocol (DAP) version: 1.48
[INFO] Session started for script: HelloWorld.plutus
Loading Plutus script: HelloWorld.plutus
Compilation succeeded.
Generated UPLC (Untyped Plutus Core):
[DEBUG] Step into -> validateInputs (Line: 30)
  - Function: validateInputs
  - Input: { Inputs: [Input1, Input2] }
[DEBUG] Exception encountered at validateInputs:
  - Error: Invalid transaction input detected!
  - Stack Trace:
    1. validateInputs (Line: 30)
    2. validateTx (Line: 25)

[INFO] Debugging session terminated.
[INFO] Disconnecting from Plutus Debug Adapter...
[INFO] Debugging environment cleaned up.

```

## 5. Architecture diagram



## 6. Conclusion

The **CardanoPyC Extension** enhances the development of Cardano blockchain applications by integrating key tools directly into the PyCharm IDE. Designed for **Haskell** and **Plutus** development, it streamlines the process of building, testing, and deploying smart contracts on the Cardano network.

The extension offers **syntax highlighting** and **code completion** for Haskell and Plutus, making code easier to read and reducing errors. It also includes integration with the **Cardano Blockchain Explorer**, allowing developers to access blockchain data, view transactions, and monitor smart contract activity directly within the IDE.

One of its standout features is **wallet management**, enabling developers to interact with Cardano wallets directly from PyCharm—viewing balances, sending transactions, and deploying smart contracts without switching tools. The extension also supports **advanced debugging** through integration with **GHCi** and **devcontainers**, allowing developers to step through code, inspect variables, and troubleshoot smart contracts in real-time.

Additionally, the extension simplifies **smart contract deployment**, allowing developers to push contracts to the Cardano testnet or mainnet with just a few clicks. Local testing within devcontainers ensures contracts perform as expected before going live.

In summary, the CardanoPyC Extension integrates essential blockchain tools into PyCharm, providing a streamlined, all-in-one environment for developing, testing, and deploying Plutus smart contracts. This cohesive workflow enhances productivity and makes it easier for developers to manage their projects, from initial coding to final deployment.