# CREDIT SCORING PROJECT
# Plutus Smart Contracts

**Version**: 1.1

**Date**: 2023-10-05

# Document Revisions

| Date | Revision | Content | Author |
|---|---|---|---|
| 2023-09-28 | 1.0 | Initial version. | Cong Le |
| 2023-10-05 | 1.1 | Separate Credit Scoring smart contract in two contracts, one for minting and another for managing the Scoring Token when updating new base score at the beginning of each month. | Cong Le |

# Contents

# 1. Introduction

This project is to calculate the credit scoring for user in our system, using both onchain and offchain data. After that, the scoring will be used in other systems, for example: to borrow money from Lending contract, etc.

There are three parts of this project:

**+ First, collect user data.**

This part is out of scope of Plutus smart contracts, it's more about data mining / data processing than smart contract. Because we need to collect all of user data, including both onchain and offchain data, then process data to figure out an appropriate data model, everything is not related to Plutus smart contracts in this step.

There are some onchain data that we can collect based on user address, for example:

- Address balance
- Staking reward
- Payment frequency (number of transactions)
- Monthly/quarterly/yearly payment (total sent, total received)

There are some offchain data that we can collect, for example:

- KYC data (to check AML status, bank credit score, …)
- Linked addresses (to check score in other networks: ETH, BTC, …)

In the first version of this project, we will only use onchain data to calculate the credit scoring for user in our system.

**+ Second, mint a new Scoring Token for user with score attached in datum.**

The user's score will be calculated, and then, if the score is greater than a threshold that we defined, we will mint a new Scoring Token for user with score attached in datum, and the Scoring Token will be sent to a manager contract only, not sending directly to user's address.

**+ Third, demonstrate how to use the Scoring Token in other systems.**

Give an example on how to use the Scoring Token, for example: use it to borrow money from Lending contract.

# 2. Architecture

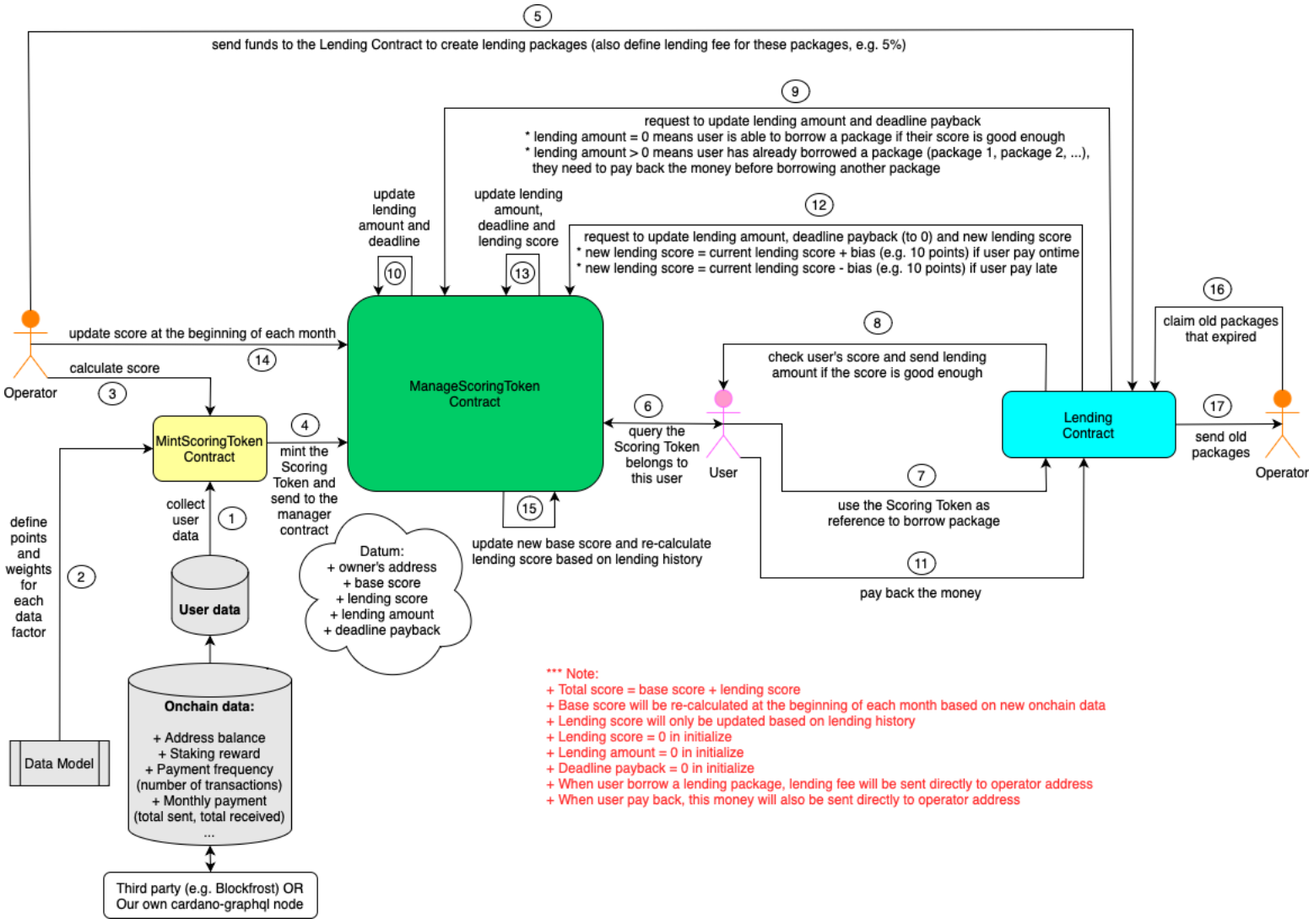


*Figure 1. All the scenarios related to Plutus Smart Contracts in the system*

## 2.1. Actors

+ Operator: only this actor has permission to calculate the score, and mint the Scoring Token for user. They also send funds to Lending contract to create some lending packages, and then claim back old (expired) packages.

+ User: this actor is able to use the Scoring Token as a reference to borrow money from Lending contract, and then pay back the money. All credit information related to user will be calculated/updated in the datum attached with the Scoring Token.

## 2.2. Scenarios

+ Step 1: collect user data (this is out of scope of Plutus smart contracts).

In the first version, we will collect these onchain data:

- Address balance (only check ADA asset)
- Staking reward
- Payment frequency (number of transactions) per month
- Monthly payment (total sent, total received)

We are able to collect onchain data by two options:

- Use third party (e.g. Blockfrost)
- Build our own cardano-graphql node

The better choice for now is using Blockfrost first, because this provider is really powerful and free, it also has all the things that we need. Later than, if we care about the availability of our system, we will think of building our own cardano-graphql node.

+ Step 2: define a data model to set points and weights for each data factor (this is out of scope of Plutus smart contracts).

Here are points and weights that I define for each factor in this version:

- Address balance:
  - 0 points: 0 <= balance < 100 ADA
  - 10 points: 100 <= balance < 1,000 ADA
  - 20 points: 1,000 <= balance < 10,000 ADA
  - 30 points: 10,000 <= balance < 100,000 ADA
  - 50 points: balance >= 100,000 ADA

- Staking reward:
  - 0 points: 0 <= reward < 1 ADA
  - 10 points: 1 <= reward < 10 ADA
  - 20 points: 10 <= reward < 100 ADA
  - 30 points: 100 <= reward < 1,000 ADA
  - 50 points: reward >= 1,000 ADA

- Payment frequency:
  - 10 points: 0 < history_count < 10 txs
  - 50 points: 10 <= history_count < 50 txs

    - 30 points: 50 <= history_count < 100 txs
    - 20 points: 100 <= history_count < 1,000 txs
    - 0 points: history_count = 0 OR history_count >= 1,000 txs

- Monthly payment:
    - 0 points: 0 <= total_sent < 100 ADA
    - 10 points: 100 <= total_sent < 1,000 ADA
    - 20 points: 1,000 <= total_sent < 10,000 ADA
    - 30 points: 10,000 <= total_sent < 100,000 ADA
    - 50 points: total_sent >= 100,000 ADA

- Weight:
    - Address balance: 25
    - Payment frequency: 25
    - Monthly payment: 25
    - Staking reward: 25

Based on that, we will have some information about min score, max score and ranges:

- Min score: 0 x 25 + 0 x 25 + 0 x 25 + 0 x 25 = 0
- Max score: 50 x 25 + 50 x 25 + 50 x 25 + 50 x 25 = 5000
- Ranges:
    - Bad: 0 -> 1000
    - Normal: 1000 -> 2000
    - Good: 2000 -> 3000
    - Very Good: 3000 -> 5000

**About “Address balance” factor**

At the moment, I just care about ADA balance (ignore tokens and NFTs). A large amount is nothing but still means something about user’s address.

**About “Staking reward” factor**

I checked ADA’s staking reward on this page: https://www.stakingrewards.com/

If I delegate to Moonstake validator, here are the staking reward corresponding to the delegated amount:

- Delegate 100 ADA → staking reward: 0.353351 ADA
- Delegate 1,000 ADA → staking reward: 3.53351 ADA
- Delegate 10,000 ADA → staking reward: 35.3351 ADA

- Delegate 100,000 ADA → staking reward: 353.351 ADA
- Delegate 1,000,000 ADA → staking reward: 3,533.51 ADA

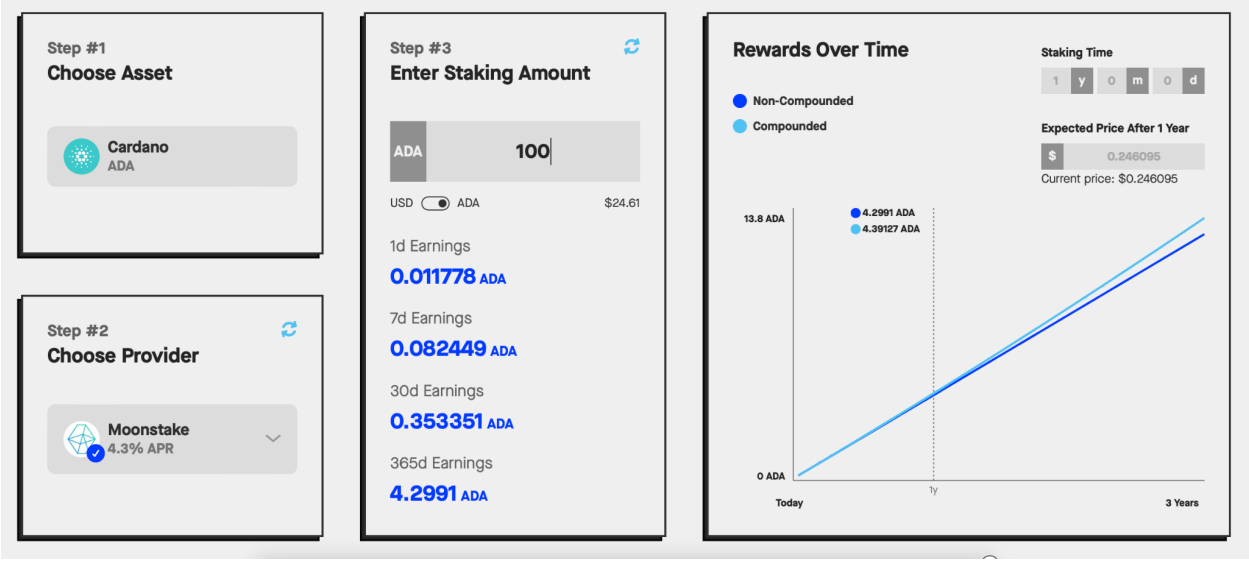


*Figure 2. ADA staking reward if user delegate to Moonstake validator*

So that's the reason why I set points for staking reward as above.

**About "Payment frequency" factor**

A small history_count means this address is not really active on the chain, and too much history_count means it might be a bot address, not a human address.

**About "Monthly payment" factor**

I use total_sent instead of total_received because:

If user want to send, they need to receive ADA before, so the meanings of total_sent might include the total_received.

If user try to cheat this factor by receiving and sending a lot, then the score of Payment frequency will be decreased.

+ Step 3: the operator interacts with MintScoringToken contract to calculate user's score based on data and model from step 1 & 2

+ Step 4: if user's score is good enough (greater than or equal a threshold that we defined), the operator will mint a new Scoring Token for user with score and some information attached in datum. The Scoring Token will be sent to ManageScoringToken contract (manager contract).

Here are some information in the datum:

- Owner's address (pubkeyhash + scripthash)
- Base score
- Lending score
- Lending amount
- Deadline payback

Total score = base score + lending score

Base score will be re-calculated at the beginning of each month based on the new onchain data.

Lending score will only be updated based on the lending history, this field is zero in initialize.

Lending amount is to track which package user is borrowing from Lending contract, this field is 0 in initialize.

Deadline payback is the end time to pay back the money to Lending contract, it is used to track a late payment.

One important thing is that the Scoring Token always stick with the ManageScoringToken contract.

+ Step 5: the operator sends funds to the Lending contract to create some lending packages.

For example:

- If user's score is greater than or equal 0 and less than 1000, user cannot borrow any package.
- If user's score is greater than or equal 1000 and less than 2000, user can borrow a package with 1,000 ADA.
- If user's score is greater than or equal 2000 and less than 3000, user can borrow a package with 2,000 ADA.
- If user's score is greater than or equal 3000, user can borrow a package with 3,000 ADA.

The Lending contract also defines a lending feerate, e.g. 5%. It means if user borrow package 1,000 ADA; they actually receive 950 ADA and 50 ADA is considered as lending fee for the operator.

+ Step 6: user will query the Scoring Token that belongs to them.

One important thing is that only the Scoring Token's owner can use the token as a reference to borrow a package from Lending contract. If you are not the Scoring Token's owner, the Lending contract will reject your transaction.

+ Step 7: user use the Scoring Token as a reference to borrow a package from Lending contract.

When user borrow a lending package, lending fee will be checked and sent directly to operator address.

+ Step 8: Lending contract will check user's score and send lending amount to user's address if the score is good enough.

+ Step 9: Lending contract will request to update lending amount and deadline payback.

The lending amount is zero means user is able to borrow a package if their score is good enough. The lending amount is greater than zero means user has already borrowed a package from Lending contract, they need to pay back the money before borrowing another package.

+ Step 10: ManageScoringToken contract sends the token to itself to update the lending amount and deadline payback in datum.

+ Step 11: user pay back the money to Lending contract.

When user pay back, this money will be checked and sent directly to operator address.

+ Step 12: Lending contract will request to update lending amount, deadline payback to zero and re-calculate the new lending score.

After user pay back the money, they will be able to borrow another package, so the lending amount, deadline payback will be reset to zero.

The lending score will be re-calculated based on the lending history. If user pay ontime, the lending score will be increased, otherwise it will be decreased.

+ Step 13: ManageScoringToken contract sends the token to itself to update the lending amount, deadline payback and new lending score in datum.

+ Step 14: at the beginning of each month, the operator will update user's score by re-collecting new onchain data, and then re-calculating the new base score.

The operator will update new base score and keep the lending amount, deadline payback (to keep the current status about whether user is borrowing a package from Lending contract or not).

But for the lending score, it depends on some conditions:

- If user is not borrowing any package, we will keep the lending score.
- If user is borrowing a package and the deadline has not reached, we will also keep the lending score.
- If user is borrowing a package and the deadline has already reached, we will re-calculate for the new lending score with some minus points.

+ Step 15: ManageScoringToken contract sends the token to itself to update new base score and new lending score (if needed).

+ Step 16: the operator claim old (expired) lending packages from Lending contract

+ Step 17: Lending contract sends old lending packages to the operator.

# 3. Expanded system

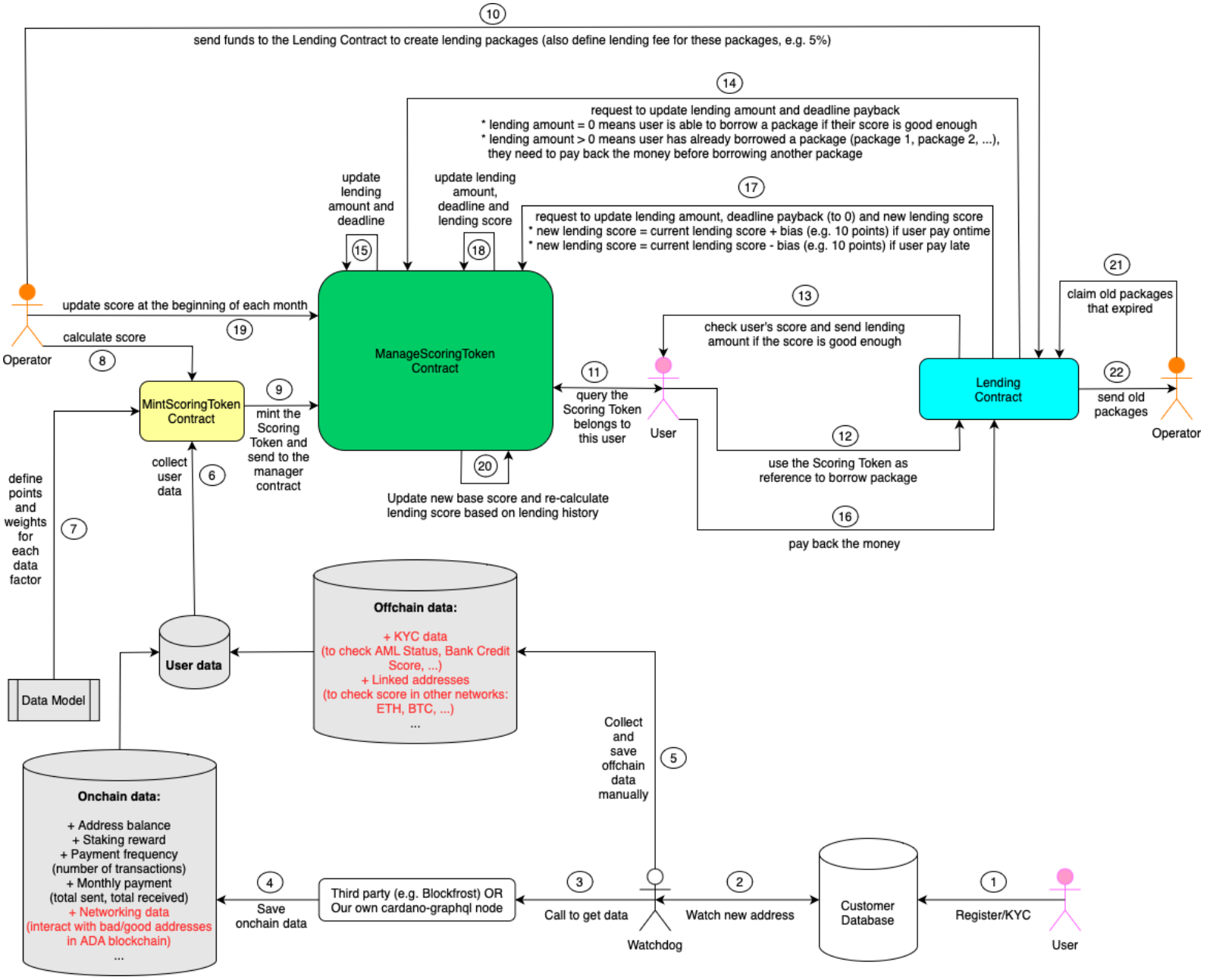


*Figure 3. All the scenarios related to Plutus Smart Contracts in the expanded system*

In future versions we will have some more features and more data to collect, including:

- Let user register KYC data into our system, and then we will use it in the offchain data.

- Add more onchain data with the networking data (data about interacting with bad/good addresses in ADA blockchain).
- Use both onchain and offchain data to calculate the score with adding some offchain data, including:
    - KYC data to check AML status, bank credit score, …
    - Linked addresses to check score in other networks: ETH, BTC,  …

Here are flows in the expanded system:

+ Step 1: user register their KYC information, the data will be saved into our Customer Database.

+ Step 2: a watchdog always watches for new registered addresses to update for our user's data.

+ Step 3: watchdog will call the third party or our own cardano-graphql node to query onchain data.

+ Step 4: Save onchain data in our onchain database.

+ Step 5: Collect and save offchain data into our offchain database.

The remaining steps are same as above in section 2.