Borrowhood: A Decentralized Microlending Platform

Technical Design Documentation – Consensus 2025

Venkateshwar Yadav, Robin Gershman, Shahrzad Masoumnia, Prisha Thakkar

May 16, 2025

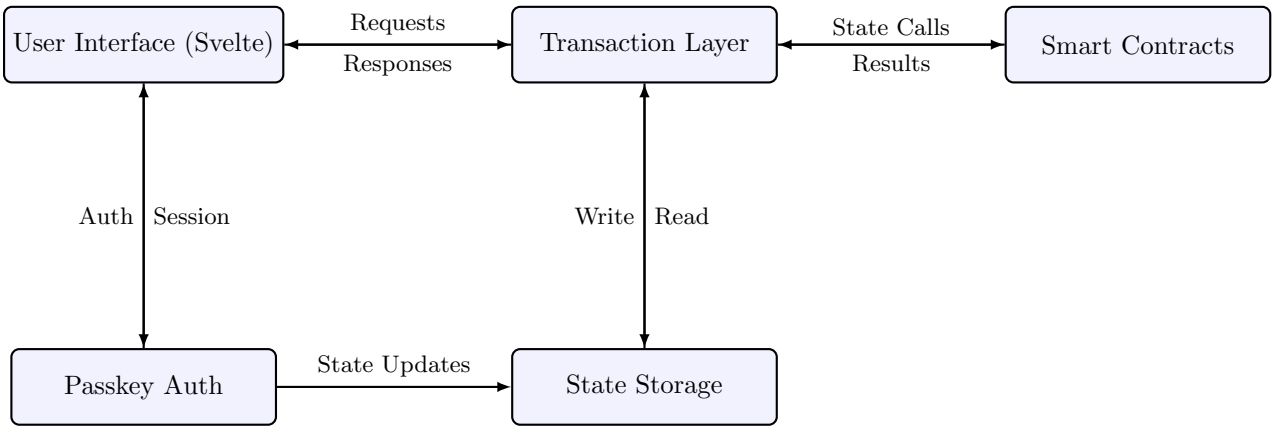
1 Overview

When we set out to build Borrowhood, we wanted to create something that hadn't really been done before - a truly user-friendly DeFi lending platform on Stellar. At its core, Borrowhood lets users lend and borrow crypto through smart contracts, all secured with passkeys instead of those annoying seed phrases everyone keeps losing.

article tikz

2 System Architecture

We structured the system around four main components that work together:



3 Major Components

3.1 Smart Contract Layer

This was probably the trickiest part to get right. We wrote our contracts in Rust using Soroban (Stellar's new smart contract platform). The contract handles:

- Token management - We started with just XLM and USDC, each with their own lending/borrowing rates
- Pool management - For holding liquidity from lenders
- Loan tracking - From request to repayment

One thing we learned quickly: Soroban's type system is great for preventing bugs, but it meant we had to be really precise about our data structures.

3.2 Blockchain Integration Layer

We spent a lot of time on this layer because it's where we hit most of our issues. It needed to:

- Handle transactions without users understanding the blockchain details
- Deal with all the weird edge cases that happen on-chain
- Recover gracefully when things go wrong

The wrapper we built handles multiple fallback approaches for when transactions don't go through - something we didn't anticipate needing at first!

3.3 Frontend Application

We chose Svelte for our frontend because it's lightweight and really fast. Building the UI was actually fun (which isn't always the case with blockchain apps):

- Dashboard shows your lending positions and outstanding loans
- Minimal steps required to lend or borrow - we aimed for 3 clicks max
- Real-time calculations so users see exactly what they'll earn or pay

The animated gradient background was just a nice touch we added to make it feel modern.

3.4 Authentication System

This is where we got to be a bit innovative. Instead of private keys (which nobody manages well), we implemented passkeys:

- Users can log in with their fingerprint or face ID
- Keys never leave the user's device, so nothing to steal
- Works across devices for the same user

This was definitely worth the implementation effort - our test users loved how easy it was.

4 How Components Interact

When a user wants to lend some XLM, here's what happens behind the scenes:

1. They select their amount and duration in the UI
2. Our transaction layer prepares the operation details
3. The user approves with their passkey (just a fingerprint tap)
4. The transaction gets sent to the Stellar network
5. The smart contract records their position and emits events
6. UI updates to show their new lending position

It sounds simple now, but getting all these pieces to communicate smoothly took way more effort than we expected.

5 Design Choices

5.1 Storage Approach

We decided to store everything on-chain rather than using a separate database. This was mainly a philosophical choice - we wanted to be truly decentralized, even if it meant some operations would be slower.

Our data is organized in typed Maps:

```
// Tokens, pools, and loans stored as Maps
env.storage().instance().set(&symbol_short!("tokens"), &tokens);
env.storage().instance().set(&symbol_short!("pools"), &Map::<u32, LendingPool>::new(env));
env.storage().instance().set(&symbol_short!("loans"), &Map::<u32, Loan>::new(env));
```

5.2 Contract State

The contract state follows a simple progression for loans:

$$\text{Requested} \rightarrow \text{Approved} \rightarrow \text{Disbursed} \rightarrow \text{Repaid (or Defaulted)}$$

We defined this as an enum in Rust:

```
#[contracttype]
#[derive(Clone, Debug, Eq, PartialEq)]
pub enum LoanStatus {
    Requested,
    Approved,
    Disbursed,
    Repaid,
    Defaulted,
}
```

5.3 Events

We emit events at key points in the loan lifecycle. These were essential for our UI to stay in sync with the blockchain state:

- `LoanCreatedEvent` - When someone requests a loan
- `LoanDisbursedEvent` - When funds get sent to a borrower
- `LoanRepaidEvent` - When a loan is fully repaid

Originally we planned more events, but these three covered most of our needs.

5.4 Passkeys Implementation

Our passkey implementation uses the Web Authentication API (WebAuthn). It was trickier than expected since browser support varies:

```
// Simplified version of our registration flow
async function register() {
  const user = prompt("Give this passkey a name");
  if (!user) return;
  try {
    const result = await account.createWallet("Borrowhood", user);
    const { keyId: kid, contractId: cid } = result;
    // Store for future authentication
```

```
    localStorage.setItem("sp:keyId", base64url(kid));
    isLoggedIn = true;
  } catch (err) {
    alert(err.message);
  }
}
```

6 Challenges and Solutions

6.1 Transaction Handling Issues

Oh boy, this was frustrating. We kept seeing `"e4.switch is not a function"` errors when trying to sign transactions. After days of debugging, we discovered it was happening because of how Stellar's SDK returns transaction objects.

Our solution was a multi-layered approach:

- Added fallback methods for transaction submission
- Created a dummy switch function when needed
- Built a more flexible transaction interface with index signatures

This felt a bit hacky, but it worked reliably across browsers and different Stellar SDK versions.

6.2 Type Safety vs. Blockchain Reality

TypeScript was both a blessing and a curse. It caught many bugs early, but sometimes the runtime blockchain objects didn't match our type definitions.

We ended up using a lot of type assertions and flexible interfaces:

```
interface Transaction {
  built?: any;
  signatures?: string[];
  [key: string]: any; // Allow any additional properties
}
```

Not ideal from a type safety perspective, but necessary for working with the blockchain.

6.3 State Synchronization

Keeping the UI in sync with blockchain state was tricky due to transaction confirmation delays. We implemented:

- Loading states for all actions
- Optimistic UI updates (show the expected result immediately)
- Background polling for confirmation

In retrospect, we should have built this with a proper state management library from the start.

7 Conclusion

Building Borrowhood taught us a ton about blockchain development. The intersection of smart contracts, user authentication, and modern UIs is challenging but rewarding when it all comes together.

Some key lessons:

- Start with resilient error handling - blockchain operations fail more often than you'd expect
- Focus on user experience - crypto doesn't have to feel complicated
- Test across many browsers - especially for WebAuthn features

Looking forward, we plan to add multi-signature support and better risk assessment models for dynamic interest rates. The foundation we've built gives us a solid platform to extend from.