Senior Front End Engineer
Arrakis Take Home Challenge

Introduction

Hey there fellow Fremen!

We're excited for you to showcase your skills.

We understand that take-home challenges can be time consuming, so we've tried to be as clear as possible to reduce ambiguity (hence the length of the document). However, in case anything is unclear, please let us know via the Telegram channel that has been setup.

Objective

Your goal is to build a small React application that allows users to deposit funds into an Arrakis vault. You'll be implementing a focused deposit flow that handles user input validation, smart contract interactions, and transaction management.

You'll be given 7 calendar days to submit the assignment.

Context: Can you tell me more about Arrakis's users?

Not all of the context below is directly related to the assignment, however we believe in giving max context.

Who are our users?
- Our users are typically founders, heads of finance and people who are the public face of a foundation.
- They are key representatives for their respective protocols or DAOs.

What are their motivations and concerns when depositing funds?
- Users are typically depositing 10-30% of the protocol/DAOs entire portfolio into an Arrakis vault.
- This can be anything from $20k to >$1M, something deposited in a single transaction.
- As you can imagine, users are anxious when moving such large amounts of funds, so it's important to prioritize robust error handling and clear feedback throughout the transaction process.

What does the high-level user journey look like?

Let’s assume that the user has already clicked on “Add Liquidity” CTA.

Key interactions required:
1. Connect wallet
2. Enter deposit amounts:
   - WETH token
   - USDC token
3. Sign WETH token approval [onchain transaction 1]
4. Sign USDC token approval [onchain transaction 2]
5. Sign add liquidity [onchain transaction 3]

What are the expectations of the assignment?

At a high-level, we will assess:
- Code architecture and organization
- React development practices (components, hooks, state management)
- TypeScript implementation
- Smart contract interactions
- Error handling and input validation
- Basic responsive design using Tailwind CSS

Below you can find detailed tables explaining our expectations.

Code Structure and Development

Area	**In Scope**	**Out of Scope**
React Components	• Functional components with hooks • Clean component hierarchy • Proper prop typing • Basic state management	• Complex state management solutions (Redux, etc) • Server-side rendering optimization
TypeScript	• Proper interface/type definitions • Type safety for contract interactions • Basic generic types	• Advanced type system features • Custom type utilities
Testing	• Basic unit tests for core functions • Component rendering tests	• End-to-end testing • Extensive test coverage • Integration tests

Web3 Integration

Area	**In Scope**	**Out of Scope**
Wallet Connection	• RainbowKit integration • Basic connection state handling • Network checking	• Custom wallet connectors • Advanced wallet features

Contract Interaction	• Token approvals • Deposit transaction • Basic error handling	• Transaction queueing • Advanced error recovery • Gas optimization
Transaction Status	• Basic loading states • Success/error feedback	• Detailed transaction tracking • Transaction history • Advanced status notifications

UI Implementation

Area	**In Scope**	**Out of Scope**
Responsive Design	• Desktop-first responsive layout • Tailwind CSS usage • Basic component scaling	• Mobile optimization • Complex responsive behaviours • Custom CSS solutions
Input Validation	• Basic amount validation • Balance checking • Input formatting	• Advanced validation patterns • Custom validation rules
Error Handling	• User feedback for common errors • Basic error states	• Detailed error reporting • Error analytics • Advanced error recovery flows

Time Expectations

- Focus on core functionality over extra features
- Document any incomplete items or assumptions
- Prioritize code quality over feature quantity

Out of Scope

- Complex state management solutions
- Advanced transaction tracking
- Detailed analytics or logging
- Mobile-specific optimizations
- Custom wallet integration
- Advanced error recovery mechanisms

Technical Requirements

You must use:
- TypeScript

We don’t have a preference on what other technologies you use, we suggest you prioritise what you are most comfortable with.

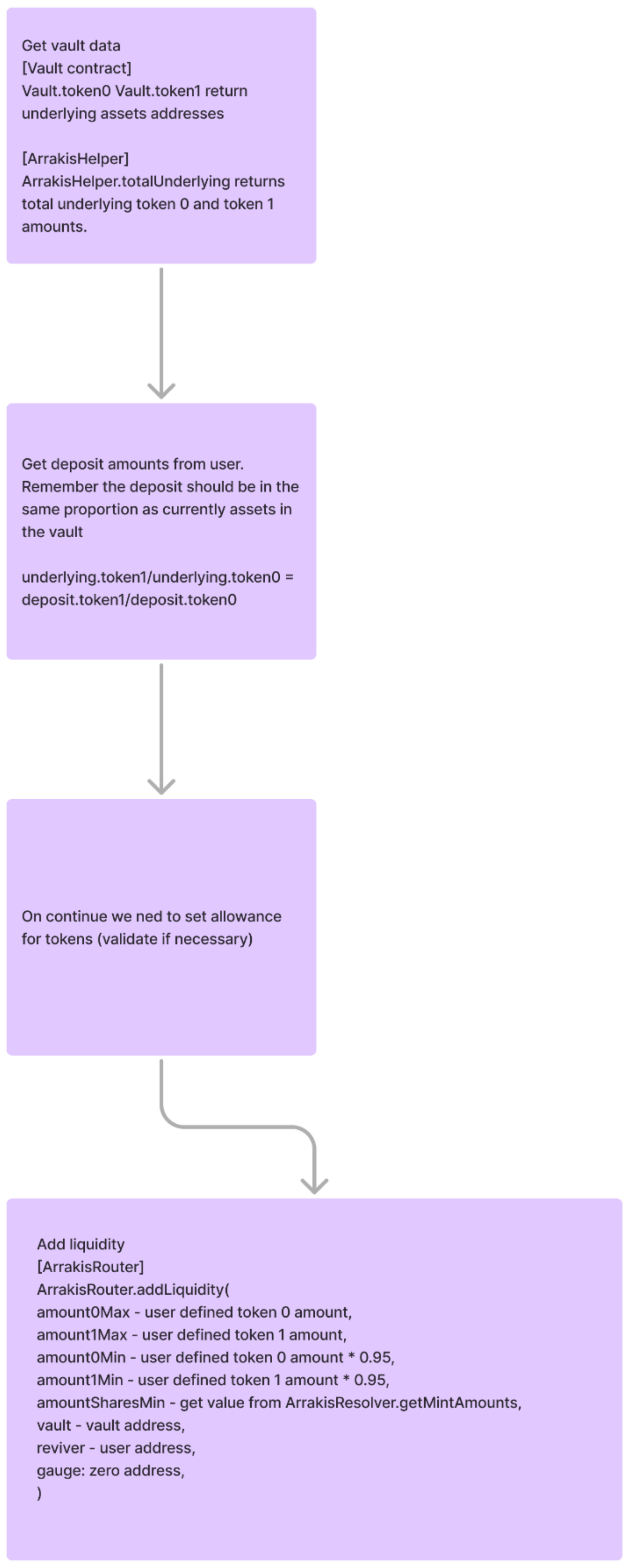
Our FE tech stack:
- React or Next.js
- Tailwind CSS
- Ethers.js or Viem or Wagmi
- RainbowKit

We suggest you use:
- a starter kit to reduce the project setup time e.g. create-nextjs-app.
- shadcn for creating basic components to save you time or any component library.

Resources

- Smart Contract Addresses:
    Arrakis helper: 0x89E4bE1F999E3a58D16096FBe405Fc2a1d7F07D6
    Arrakis router: 0x6aC8Bab8B775a03b8B72B2940251432442f61B94
    Arrakis resolver: 0x535c5fdf31477f799366df6e4899a12a801cc7b8

Vault id to interact with: 0x4ca9fb1f302b6bd8421bad9debd22198eb6ab723
It is a vault on the Arbitrum network, holding WETH and USDC.



Get vault data
[Vault contract]
Vault.token0 Vault.token1 return underlying assets addresses

[ArrakisHelper]
ArrakisHelper.totalUnderlying returns total underlying token 0 and token 1 amounts.

Get deposit amounts from user.
Remember the deposit should be in the same proportion as currently assets in the vault

underlying.token1/underlying.token0 = deposit.token1/deposit.token0

On continue we ned to set allowance for tokens (validate if necessary)

```
Add liquidity
[ArrakisRouter]
ArrakisRouter.addLiquidity(
amount0Max - user defined token 0 amount,
amount1Max - user defined token 1 amount,
amount0Min - user defined token 0 amount * 0.95,
amount1Min - user defined token 1 amount * 0.95,
amountSharesMin - get value from ArrakisResolver.getMintAmounts,
vault - vault address,
reviver - user address,
gauge: zero address,
)
```

What support can I get?

If you have any questions, you can:
- Setup a 20 min call with us (ping us on Telegram to arrange)
- Message us with questions on Telegram

We welcome you asking questions, so please don't be shy :)

Submission

Please work in the GitHub repository we've created for you and ensure your final submission includes:

1. Working application code
2. Basic test coverage
3. README.md with:
   - Setup instructions
   - Any assumptions made
   - Known limitations
   - Areas for improvement
   - Local development instructions

Please complete your work within 7 calendar days. If you need more time, let us know as soon as possible.

FAQs

Q: Do I need to implement advanced error handling and recovery?
A: No, focus on basic error handling for common scenarios (insufficient balance, rejected transactions, etc).

Q: Should I implement my own state management solution?
A: No, for example React's built-in state management (useState, useContext) is sufficient.

Q: Do I need to write extensive tests?
A: No, focus on basic unit tests for core functionality and component rendering.

End

We're super excited to see your submission!

The liquidity must flow...