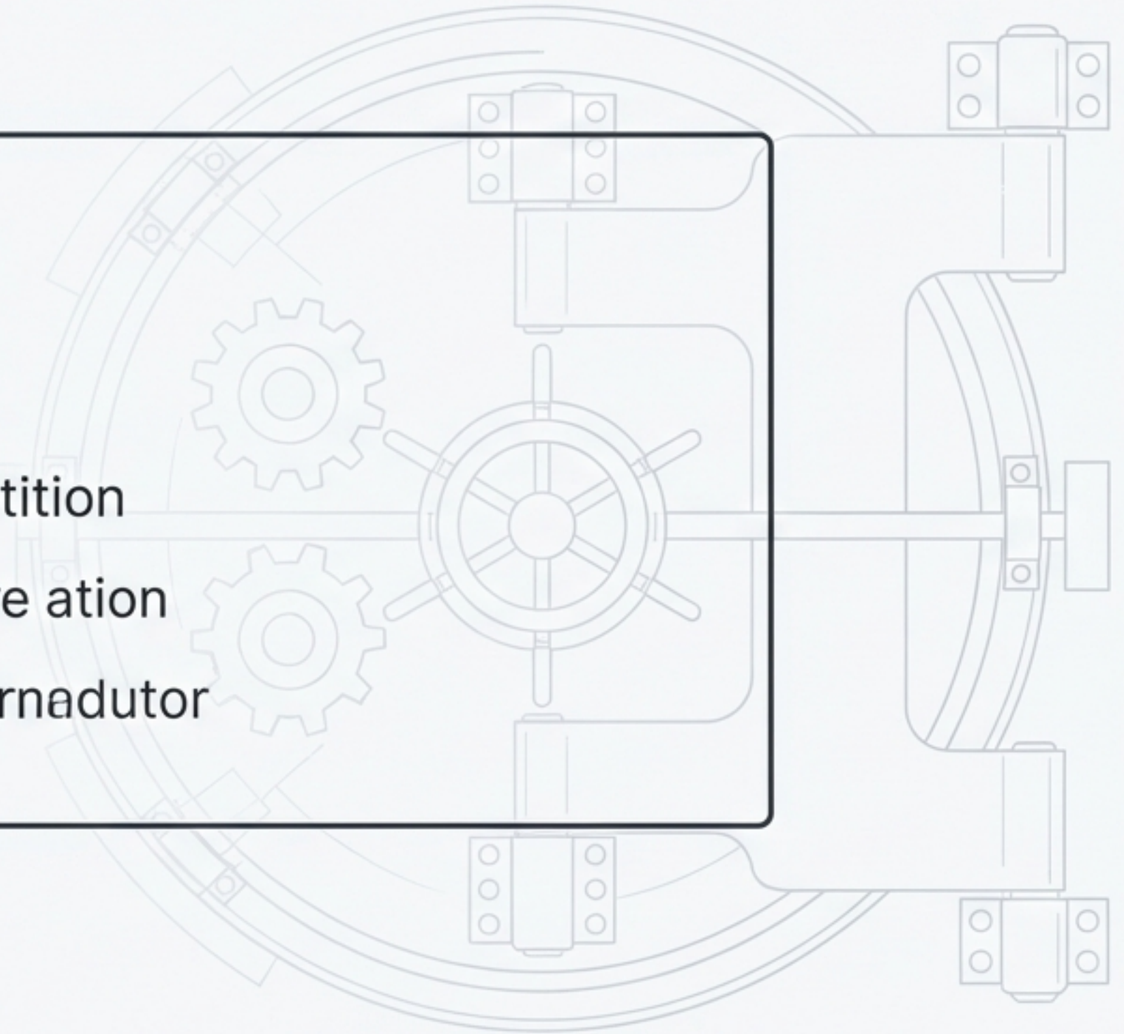
Your Mission: Build a Solana Vault

Let’s walk through the task Burhan gave you, step-by-step.

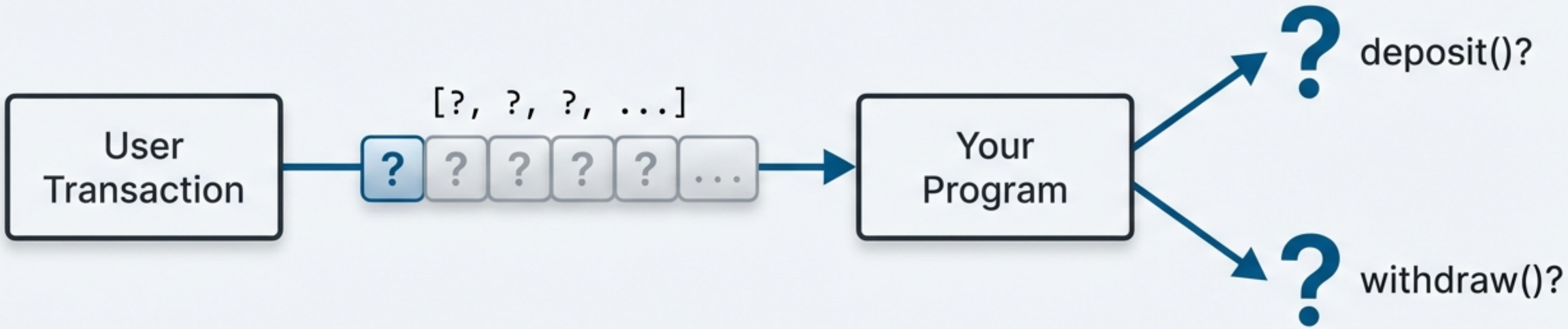


Your Assignment from Burhan:

- Build a Bank Vault ebm vault
- Build recent cient-and competition
- Forge tuilc a Lockup Configure ation
- Build a Solana to clemit vault rnadutor

The First Hurdle: How Does a Program Understand Commands?

Your Solana program will receive **raw data from** users as a **stream of bytes** (**&[u8]**`). How does it know if the user wants to **deposit** or **withdraw**?

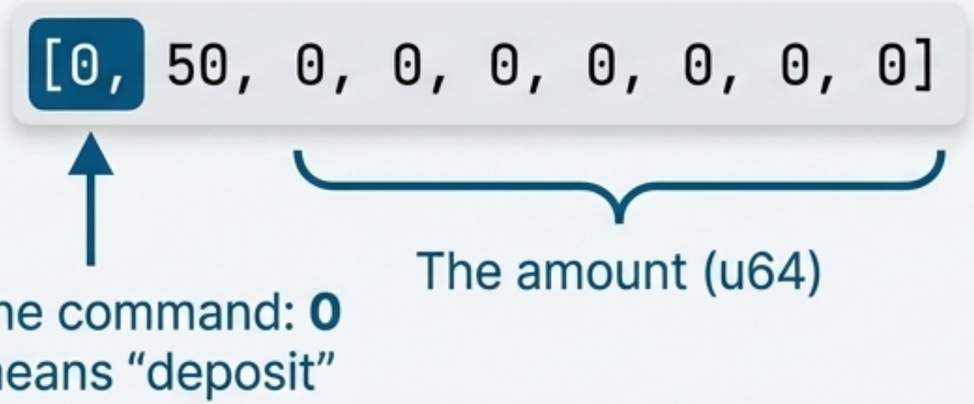


The program needs a system to interpret these raw bytes into specific actions. We'll build that system now.

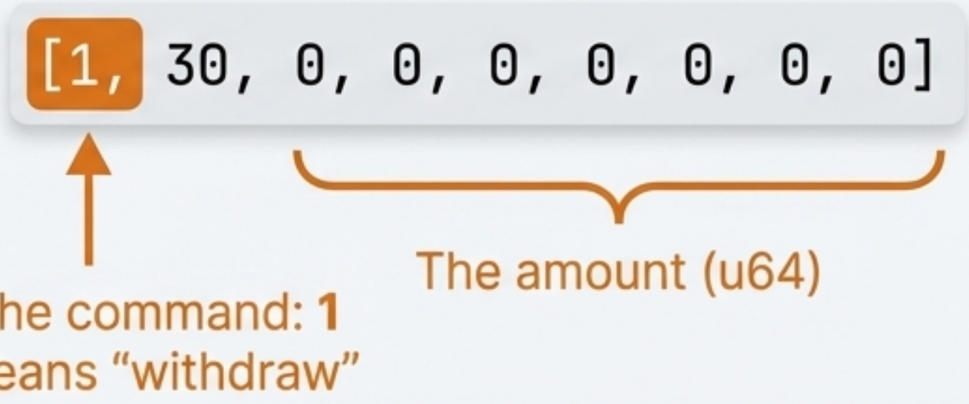
The Solution: A Simple Command System

We'll use the **first byte** of the instruction data to tell our program what to do. This first byte is called a "discriminant."

To **DEPOSIT**:
The user's instruction data will look like this:



To **WITHDRAW**:
The user's instruction data will look like this:



This pattern is fundamental. `instruction_data[0]` tells you the "what," and `instruction_data[1..]` tells you the "how much" or other details.

Your Tool for Routing Commands: The `match` Statement

Rust's `match` statement is perfect for this. It's like a powerful switch that checks a value and runs code based on which "arm" it matches.

Let's look at a simple example:

```
// A simple Rust example
let instruction_byte = 0;

match instruction_byte {
    0 => println!("This is deposit"),
    1 => println!("This is withdraw"),
    _ => println!("Unknown"), // This is the default case
}

// Output: This is deposit
```

We are checking the value of `instruction_byte`.

If the **value is** 0, run this code.

If the **value is** 1, run this code.

The underscore `_` is a wildcard. If the value is anything else, run this default code.

Let's Write the Code: The Program's Entry Point

Every Solana program has a main entry point. All instructions come here first.
Let's write the `process_instruction` function signature.

`lib.rs`

```rust
use solana_program::{
    account_info::AccountInfo, entrypoint, entrypoint::ProgramResult,
    msg, program_error::ProgramError, pubkey::Pubkey,
};

entrypoint!(process_instruction);

pub fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    // Our logic will go here
}
```

1 entrypoint!(process_instruction);

What it does: This macro tells the Solana runtime that `process_instruction` is the function to call when our program is invoked.

2 program_id: &Pubkey

What it is: The unique address (ID) of your on-chain program.

3 accounts: &[AccountInfo]

What it is: A list of every account the transaction will read from or write to. Think of it as a "bag of keys" you need for the job.

4 instruction_data: &[u8]

What it is: The raw byte data sent by the user. This contains our `0` or `1` command!

Building the Router with `match`

Inside `process_instruction`, we'll read the first byte of `instruction_data` and use `match` to call the correct function.

inside `process_instruction`

```
pub fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {

    // First, handle the case of no data at all.
    if instruction_data.is_empty() {
        return Err(ProgramError::InvalidInstructionData);
    }

    // Now, use match on the first byte
    match instruction_data[0] {
        0 => deposit(program_id, accounts, &instruction_data[1..]),
        1 => withdraw(program_id, accounts, &instruction_data[1..]),
        _ => Err(ProgramError::InvalidInstructionData),
    }
}
```

- We are matching against the **first byte** (the discriminant).
- If the first byte is `0`, we call a new function named `deposit`.
- This is called "slicing." It passes the **rest of the data** (everything *after* the first byte) to the `deposit` function. This is where the transfer amount will be.
- If the byte is not `0` or `1`, we return an error.

Creating the `deposit` and `withdraw` Functions

Our `match` statement calls `deposit` and `withdraw`, so let's create them. For now, they will just be simple placeholders that print a message.

add this below `process_instruction`

```
// Called when instruction_data[0] == 0
fn deposit(
    _program_id: &Pubkey,
    _accounts: &[AccountInfo],
    _instruction_data: &[u8], // This is the rest of the data
) -> ProgramResult {
    msg!("Instruction: Withdraw");
    // TODO: Add SOL transfer logic here later
    Ok(())
}
```

Teaching Point: Notice how these functions have the same parameters. We pass everything down from the main `process_instruction` function. The `msg!` macro is how you print logs from a Solana program.

Your Complete Program Boilerplate

Here is all the code we've written, put together in one file. This is the complete, working structure for your program's instruction router.

```rust
use solana_program::{
    account_info::AccountInfo, entrypoint, entrypoint::ProgramResult,
    msg, program_error::ProgramError, pubkey::Pubkey,
};

entrypoint!(process_instruction);

// Main entry point - ALL calls come here first
pub fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    if instruction_data.is_empty() {
        return Err(ProgramError::InvalidInstructionData);
    }
    match instruction_data[0] {
        0 => deposit(program_id, accounts, &instruction_data[1..]),
        1 => withdraw(program_id, accounts, &instruction_data[1..]),
        _ => Err(ProgramError::InvalidInstructionData),
    }
}

// Called when instruction_data[0] == 0
fn deposit(
    _program_id: &Pubkey,
    _accounts: &[AccountInfo],
    _instruction_data: &[u8],
    _instruction_data: &[u8],
) -> ProgramResult {
    msg!("Instruction: Deposit");
    Ok(())
}

// Called when instruction_data[0] == 1
fn withdraw(
    program_id: &Pubkey,
    _accounts: &[AccountInfo],
    _accounts: &[AccountInfo],
    _instruction_data: &[u8],
) -> ProgramResult {
   msg!("Instruction: Withdraw");
    Ok(())
}
```

Let's Review: How the Router Works

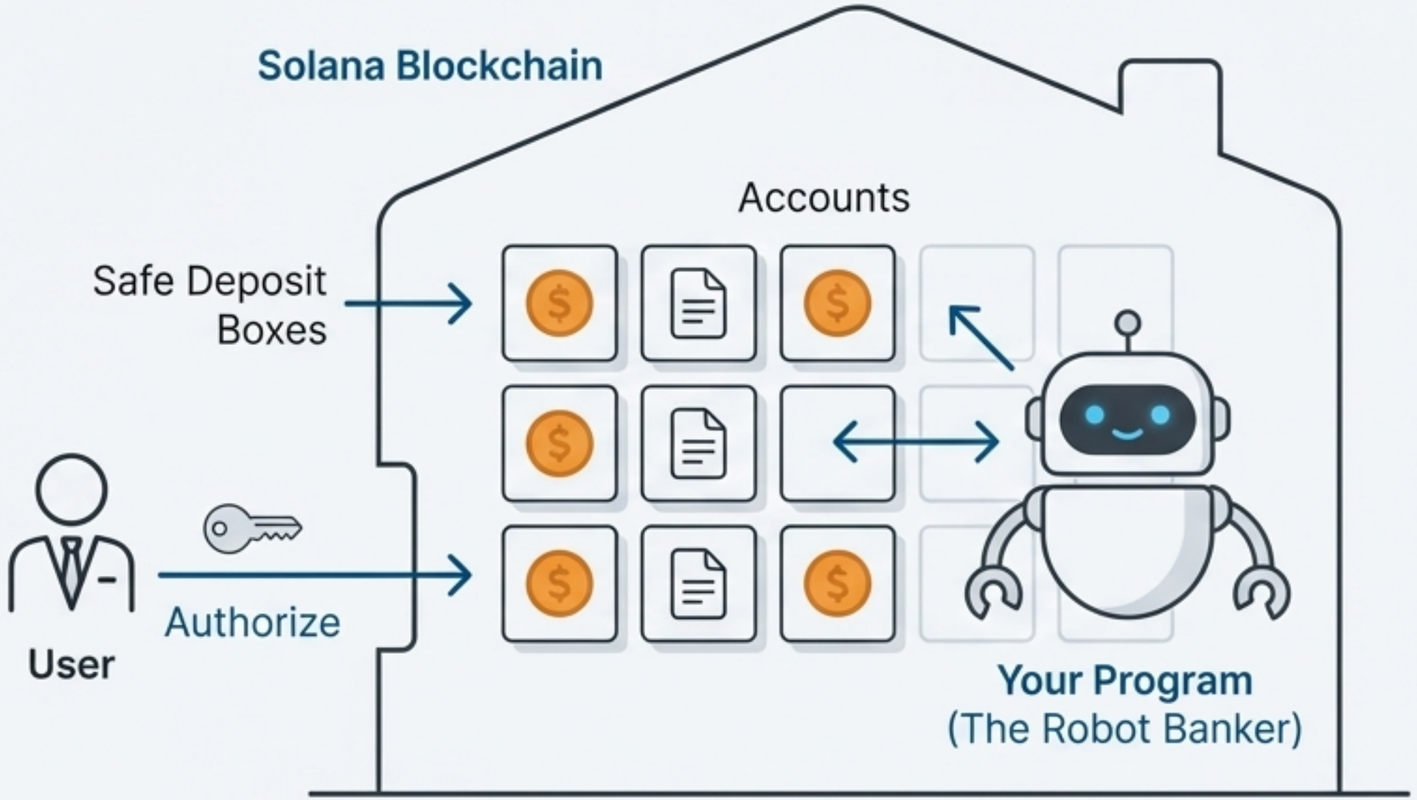
This table summarizes the logic you just built. Your program now correctly routes commands based on the first byte of data.

User Sends	First Byte	`match` Result	Function Called
`[0, 50, 0, ...]`	`0`	`matches 0 =>`	`deposit()`
`[1, 30, 0, ...]`	`1`	`matches 1 =>`	`withdraw()`
`[5, 10, 0, ...]`	`5`	`matches _ =>`	Returns **Error**
`[] (empty)`	(none)	Caught by `is_empty()` check	Returns **Error**

You've successfully completed the core routing logic. Now, let's understand the Solana architecture that makes the actual money transfers possible.

How Solana Thinks: A Mental Model

To understand how to move SOL, let's use an analogy. Imagine your program is a Robot Banker.



- **Accounts** are like deposit boxes: They hold your SOL and data.
- **The User** has the **key**: Your private key lets you authorize moving SOL from *your* account.
- **The Program** is a **robot**: It can't sign for things itself, but it can give instructions to the blockchain to move assets it controls.

Two Ways to Move Money: `invoke` vs. `invoke_signed`

The Robot Banker has two different ways to tell the blockchain to move SOL, depending on *who* owns the money.

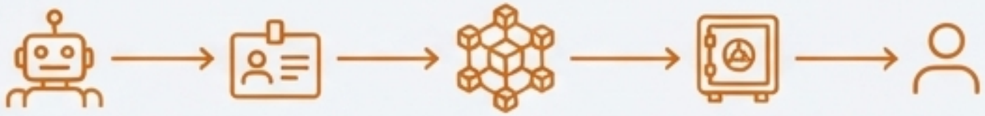
DEPOSIT (using `invoke`)

- **Goal:** User → Vault
- **Who loses money?:** The User.
- **How it works:** The User signs the transaction with their private key. Your program just says, "Hey Blockchain, the user already approved this, just do it."
- **Analogy:** You hand the banker a signed permission slip to move gold from your box to the vault. The banker just passes the slip to security.

`invoke()` = **"Pass along the user's signature."**

WITHDRAW (using `invoke_signed`)

- **Goal:** Vault → User
- **Who loses money?:** The Vault (a PDA).
- **How it works:** The Vault has no private key! The User can't sign for it. Your program must sign for it, proving it has control.
- **Analogy:** The banker needs to move gold *out* of the vault. Since the vault has no key, the banker shows its official ID badge to security, which proves it has permission.



`invoke_signed()` = **"The Program signs on behalf of its own account."**

What is a PDA? The Program's Personal Safe

The Vault account we've been talking about is a special type of account called a Program Derived Address (PDA).

Core Concept

- A PDA has a public key (an address), but it has no private key.
- A normal account is controlled by a private key (a password).
- A PDA is controlled by a program.

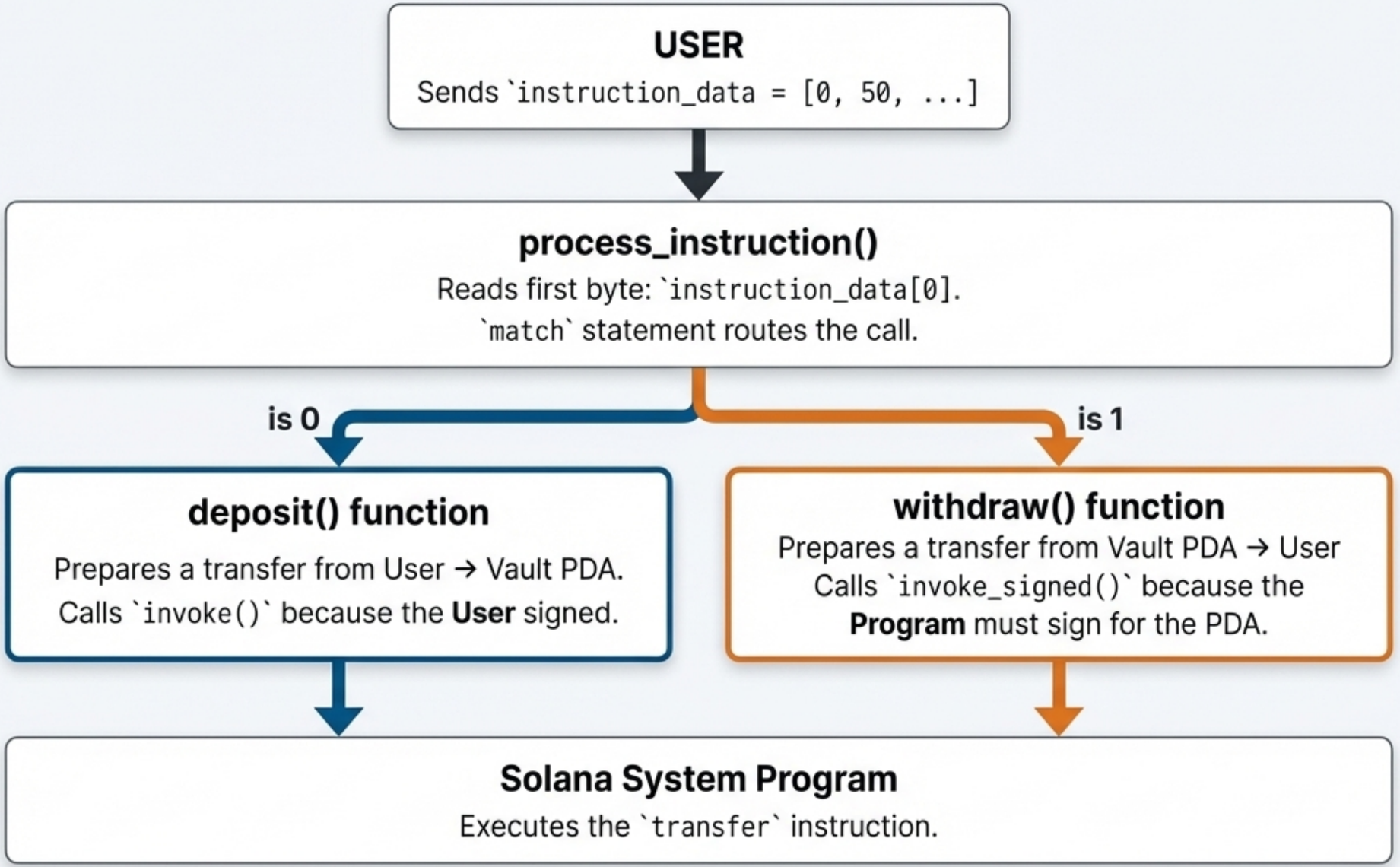
Normal Account

PDA
(Controlled by Program)

Why do we use them?

- **Security**: Since there's no private key, no human—not even the developer who wrote the code—can sign a transaction to steal the funds from the vault.
- **Trust**: It guarantees that funds can only be moved according to the rules written in your program's code.

The Complete Transaction Flow



Your Next Steps: Implementing the Transfers

You have the router. Now it's time to add the logic inside `deposit` and `withdraw` to actually move the SOL using `solana_program::system_instruction::transfer`.

In your `deposit` function, you will use `invoke`:

```
// The user signs, so we use invoke()
invoke(
    &system_instruction::transfer(user.key, vault.key, amount),
    &[user.clone(), vault.clone()],
)?;
```

In your `withdraw` function, you will use `invoke_signed`:

```
// The program signs for the PDA, so we use invoke_signed()
invoke_signed(
    &system_instruction::transfer(vault.key, user.key, amount),
    &[vault.clone(), user.clone()],
    &[&[b"vault", &[bump_seed]]], // The PDA's "virtual signature"
)?;
```

The PDA's "virtual signature"

Guidance: Focus on understanding which function to use and why. The `accounts` you'll need will be passed into your functions.

Key Takeaways & Final Check

You now have all the pieces to complete your task.

Key Concepts We've Covered:

1. **Instruction Discriminant**: The first byte (`[0]` or `[1]`) tells your program what to do.
2. **Rust** `match`: The perfect tool for routing instructions based on the discriminant.
3. `invoke()`: Used when the account losing funds is a signer (like a user depositing).
4. `invoke_signed()`: Used when the account losing funds is a PDA that the program controls (like the vault during a withdrawal).

Test Yourself

- A user sends `[0, 100, 0, 0]`. What function runs?
- A user sends `[1, 50, 0, 0]`. What function runs?
- A user sends `[3, 0, 0, 0]`. What happens?
- Why do we pass `&instruction_data[1..]` when calling the `deposit` and `withdraw` functions?