Mastering Solana Transfers: A Guide to `invoke` and `invoke_signed`

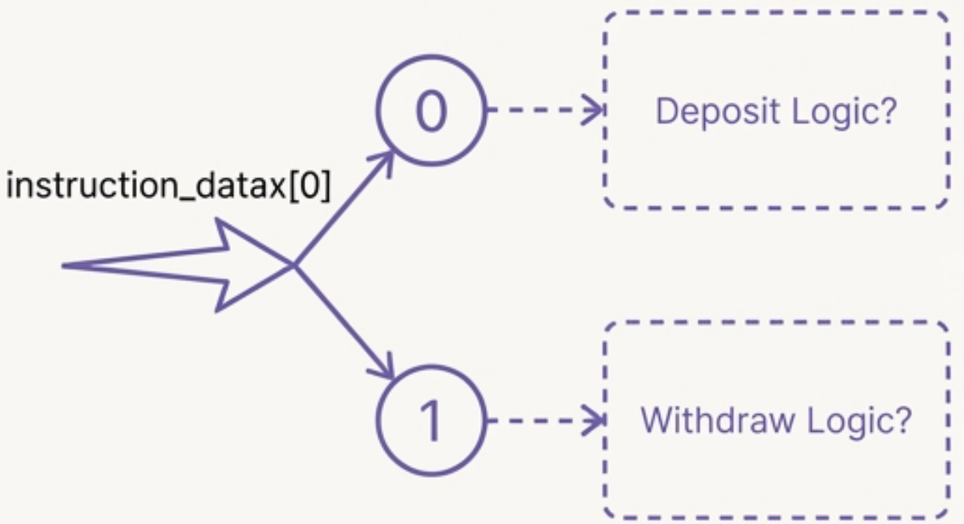
Understanding the 'Why' Behind Your Vault Program's Code

Your Starting Point

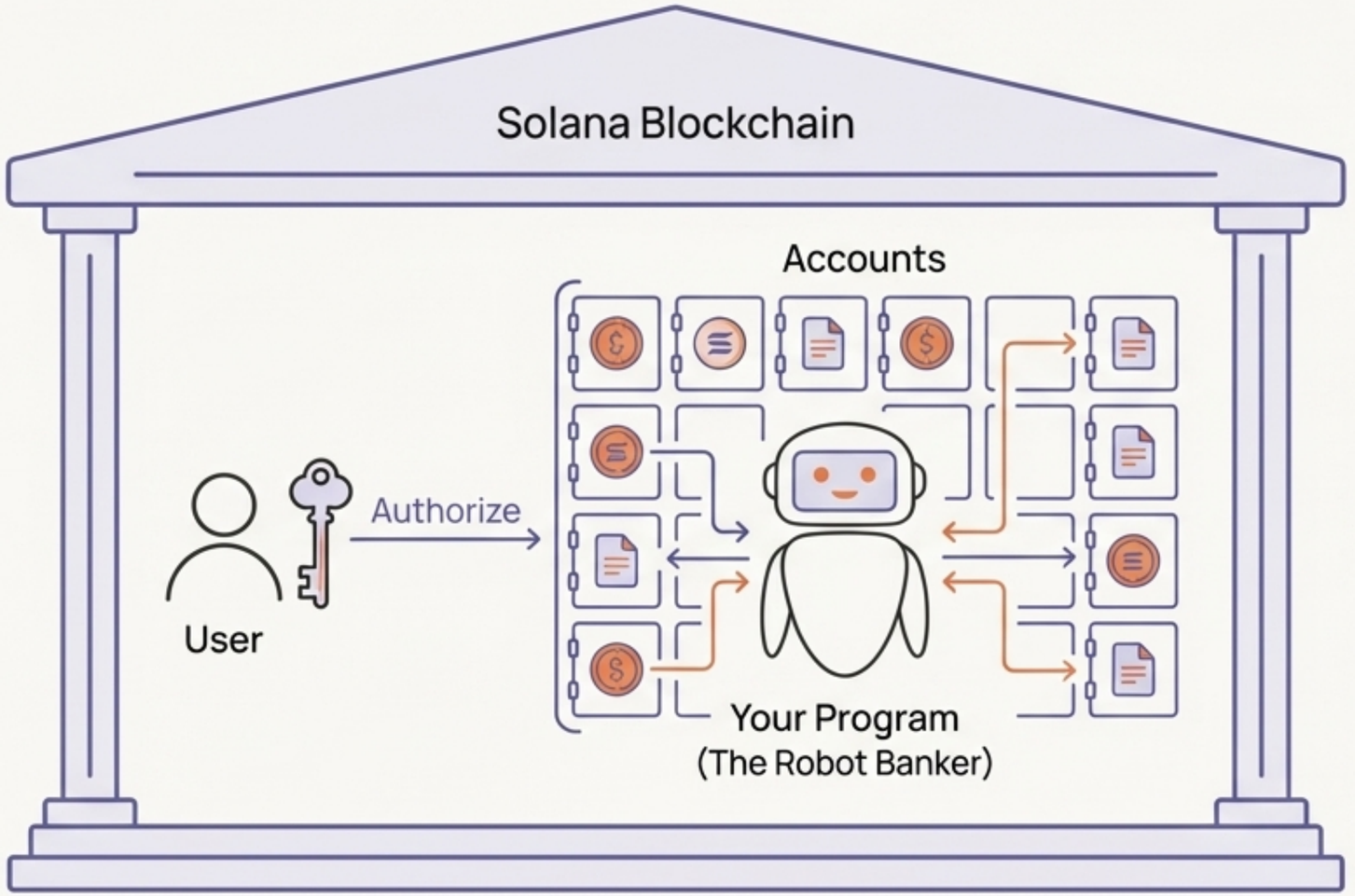
```rust
entrypoint!(process_instruction);

pub fn process_instruction(...) {
    match instruction_datax[0] {
        0 => deposit(program_id, accounts, &instruction_datax[1..]),
        1 => withdraw(program_id, accounts, &instruction_datax[1..]),
        _ => Err(ProgramError::InvalidInstructionData)
    }
}

fn deposit(...) -> ProgramResult {
    msg!("Deposit");
    // TODO: Add Functionality
    Ok(())
}

fn withdraw(...) -> ProgramResult {
    msg!("Withdraw");
    // TODO...
    Ok(())
}
```

Our Goal

Fill in these blanks. We will add the logic to securely transfer SOL, step-by-step, focusing on the two most critical functions in Solana native programming.



How Solana Thinks: A Mental Model



- **The Solana Blockchain**: is a secure bank building.
- **Accounts**: are safe deposit boxes inside the bank. They hold SOL and data.
- **The User**: is a customer with a private key to their personal box.
- **Your Program**: is a Robot Banker. It cannot sign for things itself, but it can give instructions to the blockchain to move assets it controls.

The Core Problem: How Does a Program “Own” Money?

The Program’s Personal Safe: The PDA

Normal Account

Controlled by a private key (a password).

VS.

Program Derived Address (PDA)

Controlled exclusively by the Program. **No private key exists.**

Why we use them:

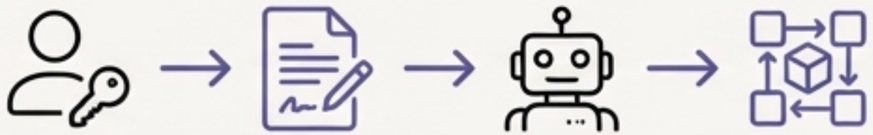
- **Security**: Since there’s no private key, no human—not even the developer who wrote the code—can sign a transaction to steal the funds from the vault.
- **Trust**: It guarantees that funds can *only* be moved according to the rules written in your program's code.

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`)

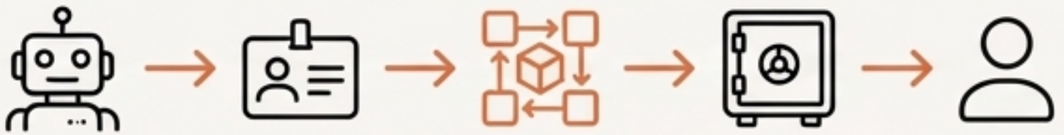
- **Goal:** User → Vault
- **Who loses money?:** The User.
- **How it works:** The User signs the transaction with their private key. Your program just says, "Hey Blockchain, the user already approved this, just do it."
- **Analogy:** You hand the banker a signed permission slip to move gold from your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

Code Deep Dive: Implementing the Deposit

Let's add the transfer logic to your `deposit` function. The key here is that the User is the one sending SOL, so they've already provided the signature we need.

```rust
// In deposit()
use solana_program:::{program::invoke, system_instruction,
    account_info::next_account_info};

// 1. Get the accounts passed in. Order matters.
let accounts_iter = &mut accounts.iter();
let user_account = next_account_info(accounts_iter)?;
let vault_account = next_account_info(accounts_iter)?;
// The amount would be parsed from _instruction_data
let amount = u64::from_le_bytes(_instruction_data.try_into().unwrap());

// 2. Define the transfer instruction using the System Program.
let transfer_instruction = system_instruction::transfer(
    user_account.key,   // From: The user is sending.
    vault_account.key,  // To: The program's vault PDA.
    amount,
);

// 3. "Invoke" the instruction, passing along the user's signature.
invoke(
    &transfer_instruction,
    &[user_account.clone(), vault_account.clone()], // Accounts involved
)?;
```

We use `invoke` because `user_account` is the one losing lamports, and the user has already signed the transaction, providing the necessary authority. We are simply forwarding **that authority** to the System Program.

Code Deep Dive: Implementing the Withdraw

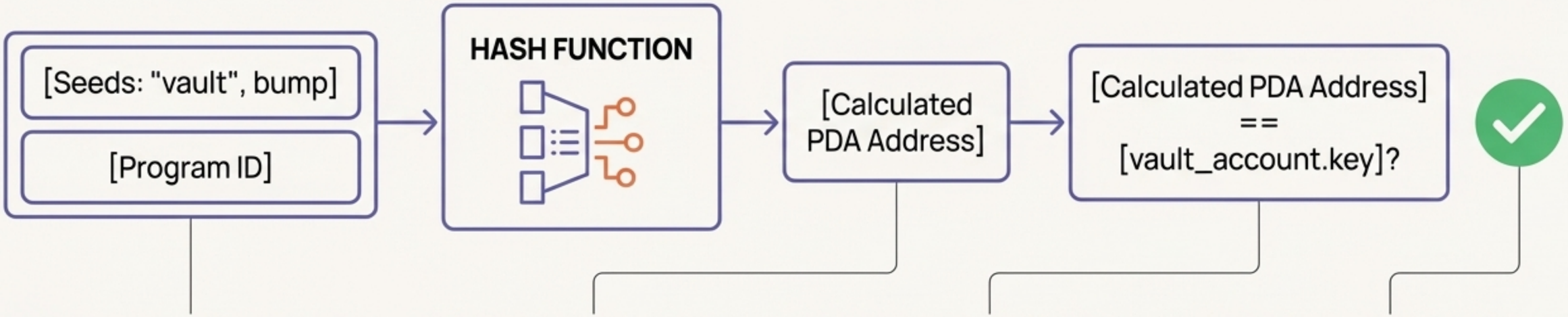
Now for the withdrawal. The Vault PDA is paying, so it needs to ‘sign’. This is where `invoke_signed` and the PDA’s seeds come into play.

```
// In withdraw()
use solana_program::{program::invoke_signed, system_instruction,
    account_info::next_account_info};

// 1. Create the transfer instruction. Notice the direction is reversed.
let transfer_instruction = system_instruction::transfer(
    vault_account.key, // From: The vault PDA is sending.
    user_account.key,  // To: The user is receiving.
    amount,
);

// 2. Define the PDA seeds. This is the recipe to prove ownership.
// The bump_seed would be passed in or derived.
let seeds = &[b"vault", &[bump_seed]];

// 3. "Invoke" the instruction WITH the PDA's "virtual signature".
invoke_signed(
    &transfer_instruction,
    &[vault_account.clone(), user_account.clone()],
    &[seeds], // <-- THE VIRTUAL SIGNATURE
)?;
```

This is the PDA's 'ID Badge'. These 'seeds' are the recipe that proves to the Solana runtime that *your program* has the authority to sign for the `vault_account`.

Deconstructing the PDA ‘Signature’

The array of bytes `&[&[b"vault", &[bump_seed]]]` isn’t a cryptographic signature. It’s a recipe.



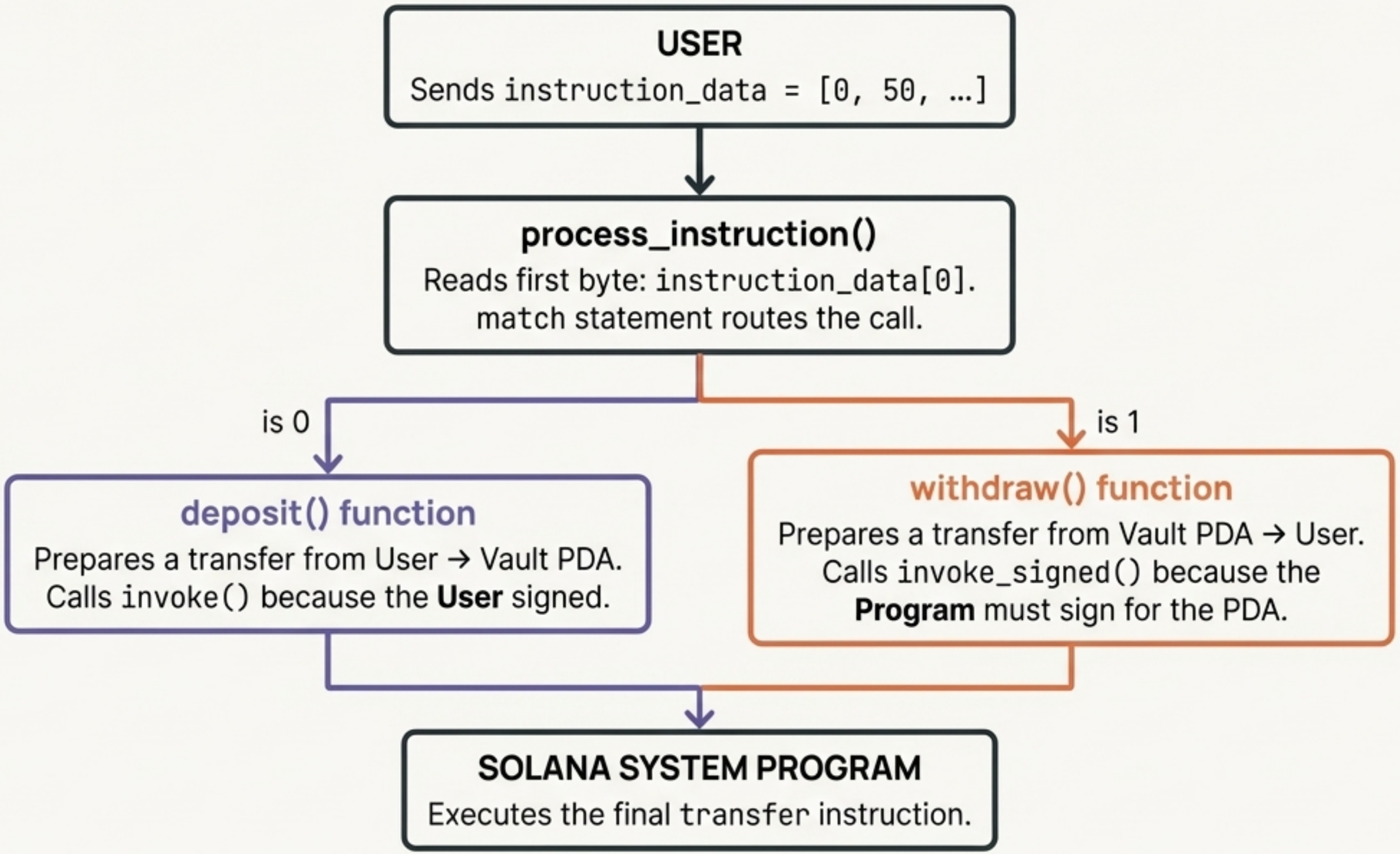
1. When your program calls `invoke_signed`, it passes these seeds to the Solana runtime.
2. The runtime uses the seeds and your program's ID to re-calculate the PDA address on the fly.
3. It then compares this calculated address to the address of the account that needs to sign (the `vault_account`).
4. If the addresses match, the signature is considered valid, and the transfer is authorized.

Your Complete Program Logic

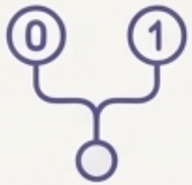
```rust
use solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint,
    entrypoint::ProgramResult,
    msg,
    program_error::ProgramError,
    pubkey::Pubkey,
    system_instruction,
};

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),
    }
}
```

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

// Called when instruction_data[0] == 1
fn withdraw(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    _instruction_data: &[u8],
) -> ProgramResult {
    msg!("Instruction: Withdraw");
    let accounts_iter = &mut accounts.iter();

    let user_account = next_account_info(accounts_iter)?;
    let vault_account = next_account_info(accounts_iter)?;

    // The bump_seed would be passed in or derived.
    let bump_seed = 254; // Placeholder for bump seed.
    let seeds = &[[b"vault", &[bump_seed]];

    let transfer_instruction = system_instruction::transfer(
        vault_account.key,
        user_account.key,
        5000000000, // 5 SOL
    );

    solana_program::program::invoke_signed(
        &transfer_instruction,
        &[vault_account.clone(), user_account.clone()],
        &[seeds], // <-- THE VIRTUAL SIGNATURE
    )?;
    Ok(())
}
```

Reviewing the Full Transaction Flow



Final Check: You're Ready



1. Instruction Routing: The first byte of `instruction_data` (`0` or `1`) is the command that tells your program what to do via a `match` statement.

2. `invoke()`: Use this when the account losing funds is a signer in the transaction (like a user depositing). You are **passing along* an existing signature.

3. PDA: A program-controlled account with no private key, used to securely hold assets on behalf of your program.

4. `invoke_signed()`: Use this when a PDA is the account losing funds (like a vault withdrawal). You are *providing* a virtual signature via seeds to prove your program's authority.

Your **Elevator Pitch** for Burhan

You can now explain this confidently. Here's how.

Why do you need a PDA?

> "Because programs can't hold SOL directly or have private keys. A PDA is a secure vault that only the program can control. It guarantees funds are managed by the code's rules, preventing theft."

Why invoke for the deposit?

> "Because the user is sending their own SOL and signs the transaction with their private key. `invoke` just forwards that existing, valid signature to the System Program to authorize the transfer."

Why invoke_signed for the withdraw?

> "Because the Vault PDA is sending the SOL. It has no private key, so it can't sign for itself. The program must "sign" for it using seeds as proof of ownership, which is what `invoke_signed` is for."

Practice saying these out loud. This will build your confidence for your meeting.