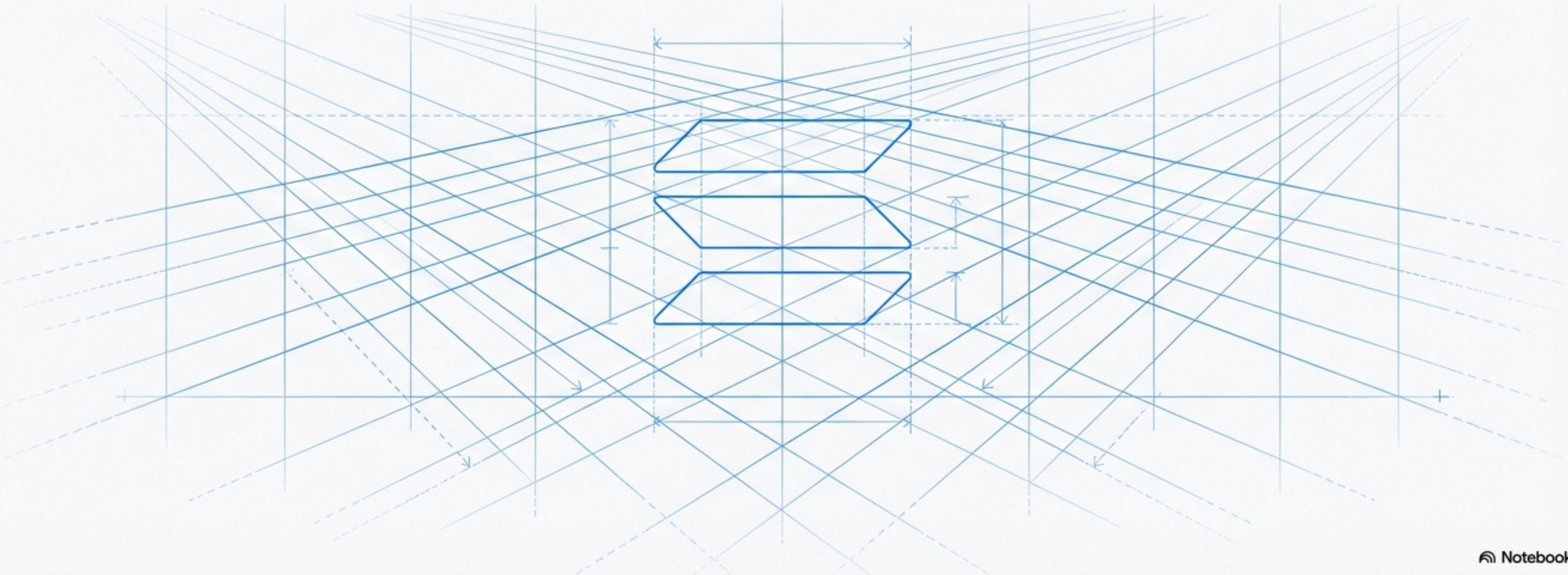
The Solana Native Blueprint

A Step-by-Step Guide from Core Concepts to Working Code



You Know Rust. But Solana Feels Alien.

The Confusion

Why are programs stateless?

What *really* is a PDA?

When do I use `invoke` vs. `invoke_signed`?

How do I actually read the `accounts` array?

The Path Forward

1. **The Mental Model:** The unshakable “why.”

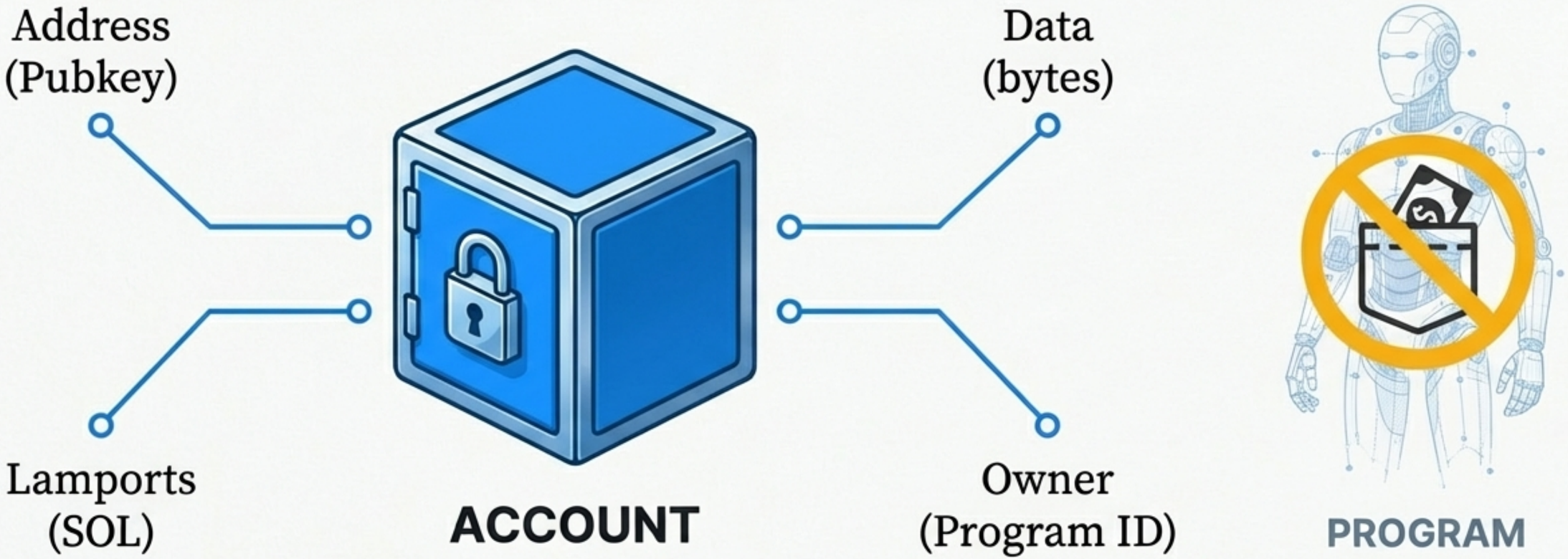
2. **The Execution Flow:** What happens, step-by-step.

3. **The Code:** The final “how,” now fully understood.

```
let account_infe = next_account_info(iter)?; // Clear access
let aeccount_info =
    next_account_info(iter, acount_info_iter);
}
```

Confidence & Mastery

The One Unbreakable Rule of Solana

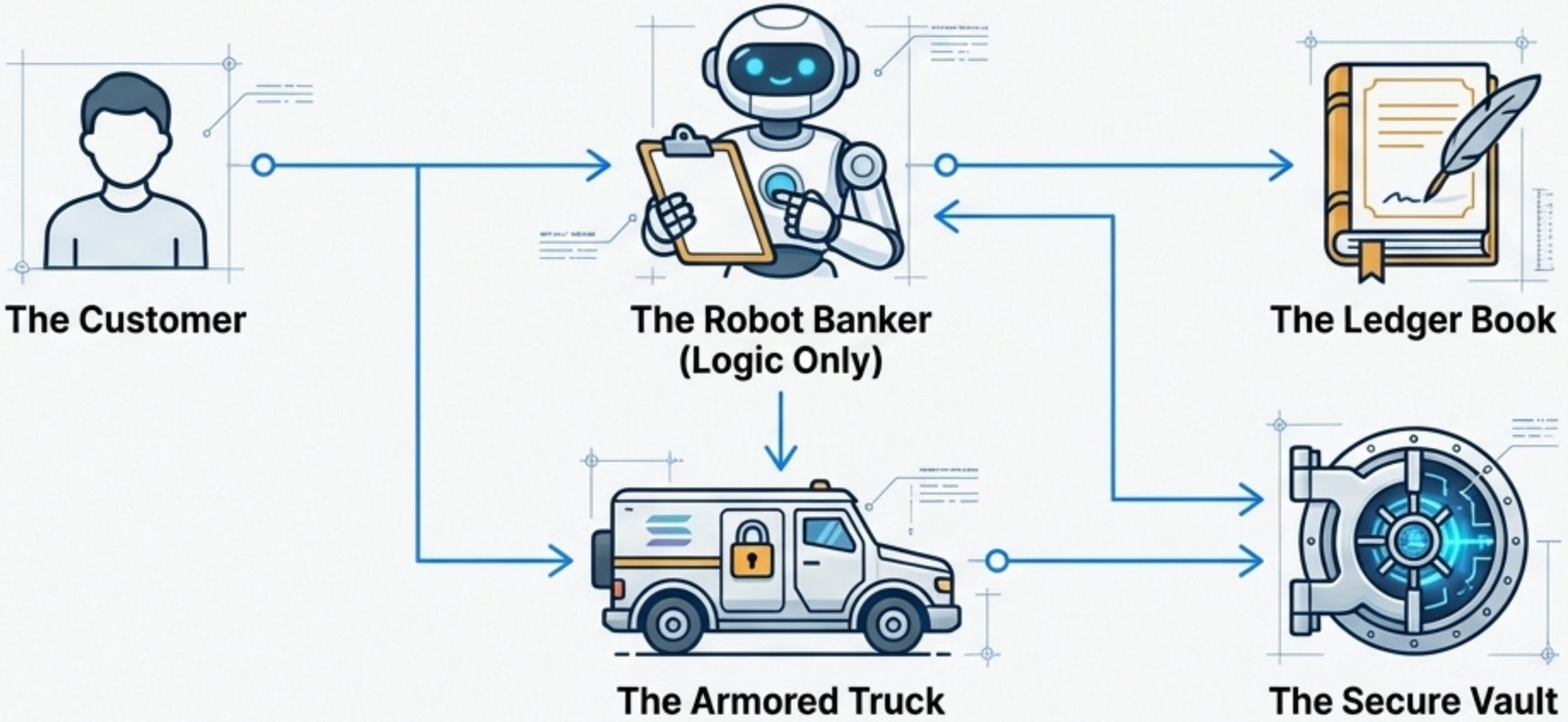


Programs are stateless logic. They are robots that can’t hold money or data.

All state and all SOL live inside accounts.

This is the most important rule to understand.

How Solana Actually Works: The Robot Banker



Your program is just the banker giving instructions. It never touches the money. It tells the Armored Truck (System Program) how to move funds between the Customer's wallet and the Vault, and it updates its Ledger (State Account).

What is a PDA? (The Program's Magic Key)

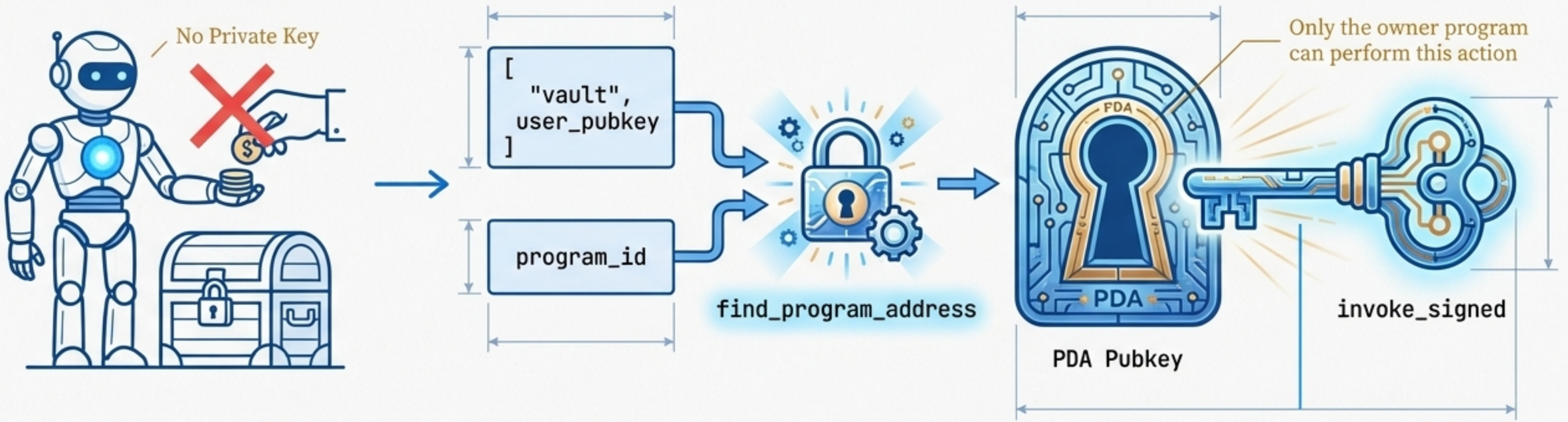
The Problem

Programs have no private keys, so they can't "sign" to authorize sending money from a vault they control.

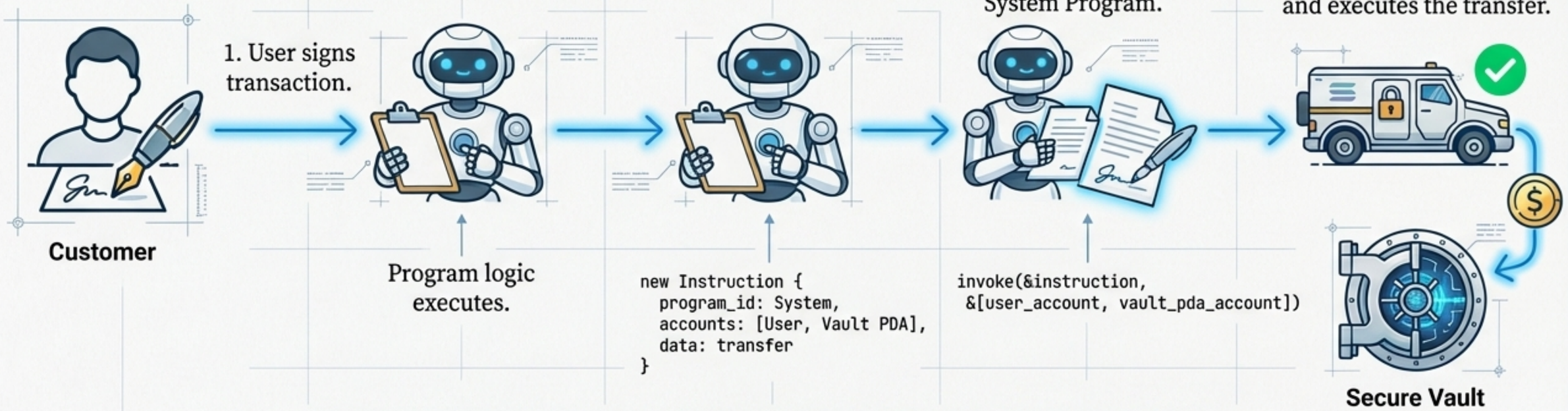
The Solution

A **PDA (Program Derived Address).**

- An address with **NO private key.**
- **Deterministically** derived from `seeds` + `program_id`.
- Crucially: **Only its owner program can authorize transactions from it.**



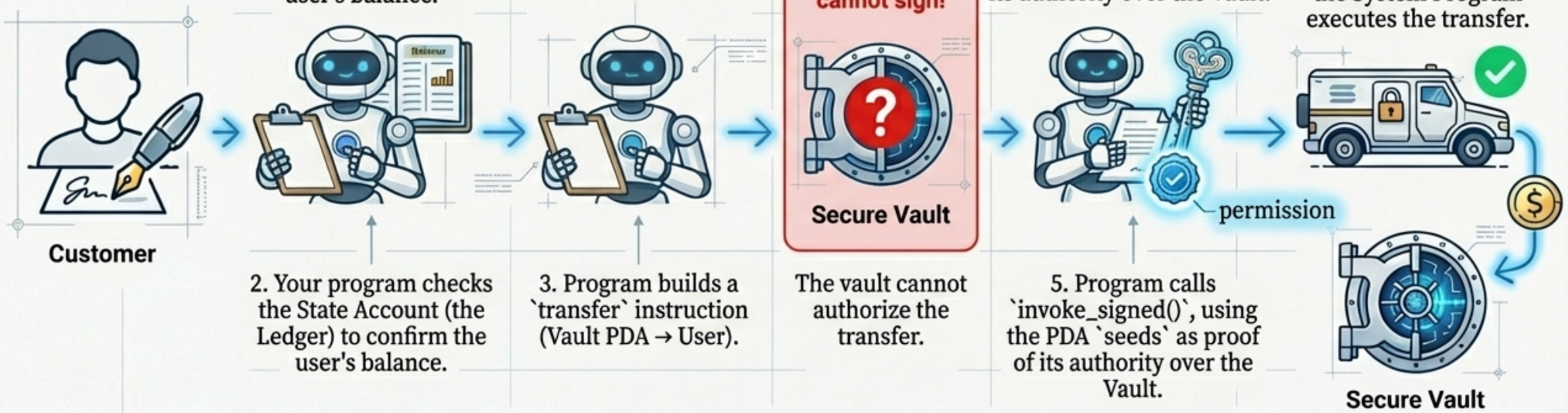
The Deposit Flow: Anatomy of an `invoke()`



Key Takeaway

Use `invoke()` when the account paying (the User) has already signed the transaction. The program is just passing the permission along.

The Withdraw Flow: Anatomy of an `invoke_signed()`



Key Takeaway: Use **`invoke_signed()`** when a PDA is paying. The program uses the `seeds` to generate a "virtual signature" on the PDA's behalf.

The Program's Front Door

The Entrypoint

```
// in lib.rs
entrypoint!(process_instruction);
```

The one and only door into your program. All calls from the Solana runtime start here.

The Dispatcher

```
// in process_instruction function...
match instruction_data[0] {
    0 => deposit(...),
    1 => withdraw(...),
    _ => Err(ProgramError::InvalidInstructionData),
}
```

The receptionist. It reads the first byte of the instruction and routes the call to the correct function. `0` for Deposit, `1` for Withdraw.

Code in Focus: The `deposit` Function

```rust
// 1. Get accounts from the array
let accounts_iter = &mut accounts.iter();
let user = next_account_info(accounts_iter)?;
let vault_pda = next_account_info(accounts_iter)?;
let system_program = next_account_info(accounts_iter)?;
// ... (also extract amount from instruction_data)

// 2. Build the transfer instruction
let ix = system_instruction::transfer(user.key, vault_pda.key, amount);

// 3. Execute with invoke()
invoke(&ix, &[user.clone(), vault_pda.clone()])?;
```

1 Reading the accounts provided by the user in the order they were sent.

2 Creating the instruction for the "Armored Truck" (System Program).

3 Using `invoke` because `user` is the payer and already signed the transaction.

Code in Focus: The `withdraw` Function

```
// 1. Get accounts & recreate PDA seeds for signing
let accounts_iter = &mut accounts.iter();
let user = next_account_info(accounts_iter)?;
let vault_pda = next_account_info(accounts_iter)?;
// ...
let (expected_pda, bump_seed) = Pubkey::find_program_address
    (&[b"vault",user.key.as_ref()], program_id);
let seeds = &[&b"vault"[..], user.key.as_ref(), &[bump_seed]];

// 2. Build the transfer instruction
let ix = system_instruction::transfer(vault_pda.key, user.key,
    amount);
// 3. Execute with invoke_signed()
invoke_signed(&ix, &[vault_pda.clone(), user.clone()],
    &&seeds[..]])?;
```

1

We must prove we own the PDA by recreating its "recipe"—the seeds.

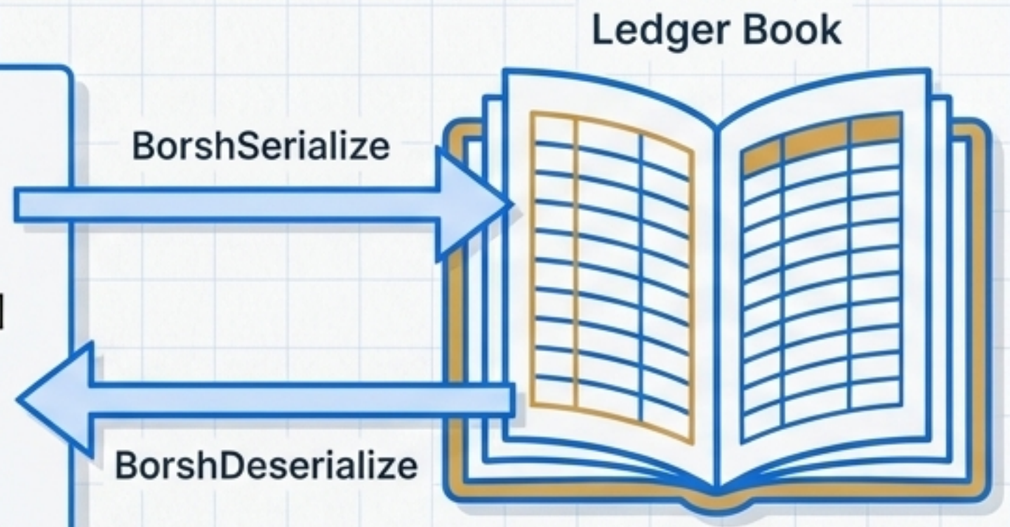
2

Using `invoke_signed` because the `vault_pda` is the payer and has no private key. The `seeds` act as its signature.

The Ledger: Storing User Balances in a State Account

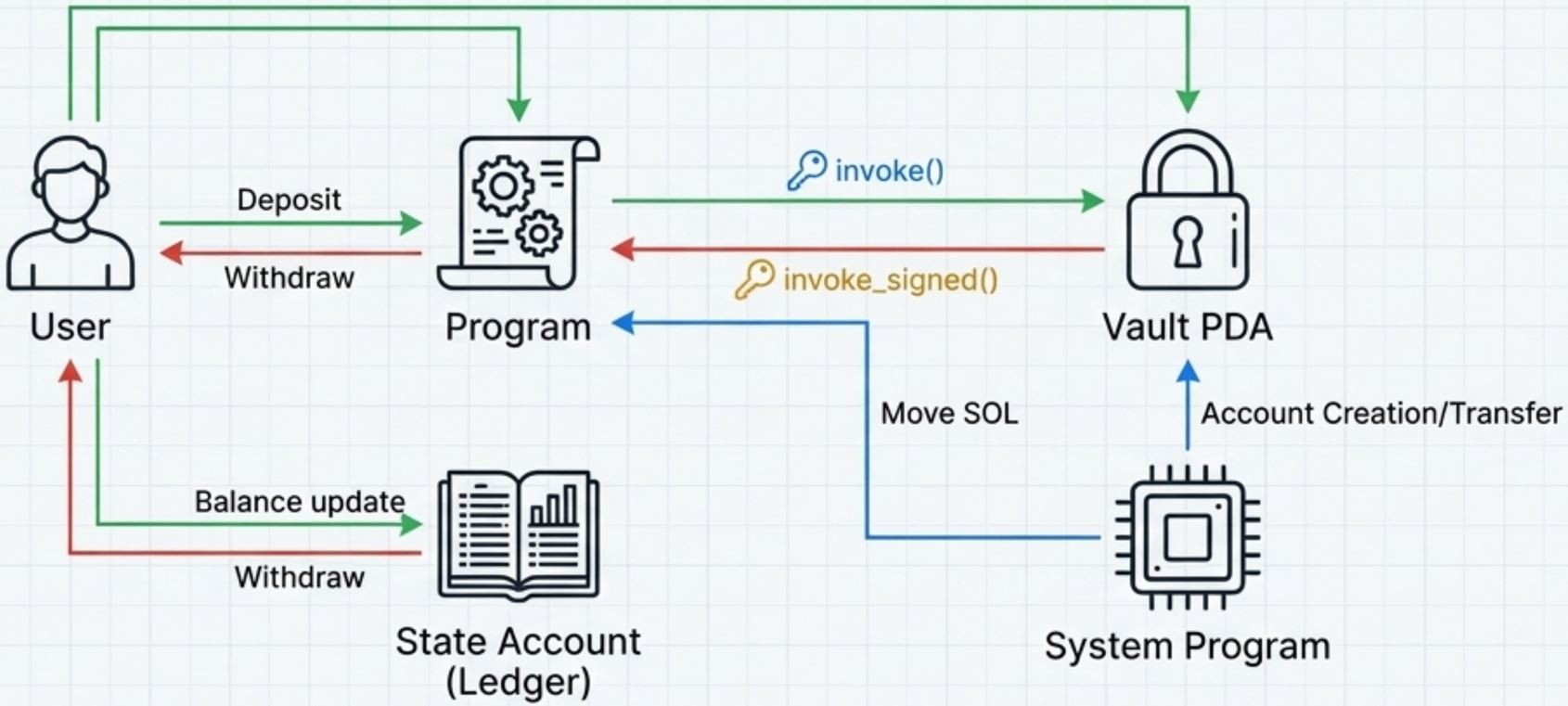
```rust
use borsh::{BorshSerialize, BorshDeserialize};
use solana_program::pubkey::Pubkey;

#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub struct UserState {
    pub user_pubkey: Pubkey,
    pub sol_deposit_amount: u64,
}
```



- This struct defines the "schema" for our program's memory.
- Each user gets their own State Account, owned by our program, to store an instance of this struct.
- `BorshSerialize` / `Deserialize` is how we write this struct to (and read from) the raw data bytes of the account.
- In `deposit()` we deserialize, `+= amount`, and serialize.
- In `withdraw()` we deserialize, check balance, `-= amount`, and serialize.

The Complete Blueprint



Your final diagram connects all the concepts: User actions trigger program logic, which uses the System Program to move SOL between accounts, all while reading and writing to a separate State account to track balances.

Passing the Test: Explaining Your Work to Burhan

Why do you need a PDA for the vault?

Because the program needs to control the vault, but programs can't have private keys. A PDA is a secure address that only our program can sign for using `invoke_signed`.

Explain the difference between `**invoke**` in **deposit** and `**invoke_signed**` in **withdraw.**

In **deposit**, the user sends their own SOL, so they sign the transaction and we use `invoke` to pass that permission. In **withdraw**, the program's PDA vault sends the SOL. Since it has no key, the program signs on its behalf using `invoke_signed` and its seeds as proof of authority.

Where is the SOL actually stored?

The SOL is stored in the lamport balance of the Vault PDA account. The program itself is stateless and holds no SOL. The State Account only stores the *record* of the user's balance, not the actual SOL.

Your Path Forward

Your Immediate Tasks

1. Complete the `withdraw` portion of your `draw.io` diagram.

2. Write the full `lib.rs` and `state.rs` code.

3. Compile and prepare to test your program.

Further Learning

- Solana Cookbook: PDAs
- Solana Cookbook: Cross-Program Invocations
- **Next Lesson:** Writing client-side tests in Rust to interact with your program.

You’ve got this.