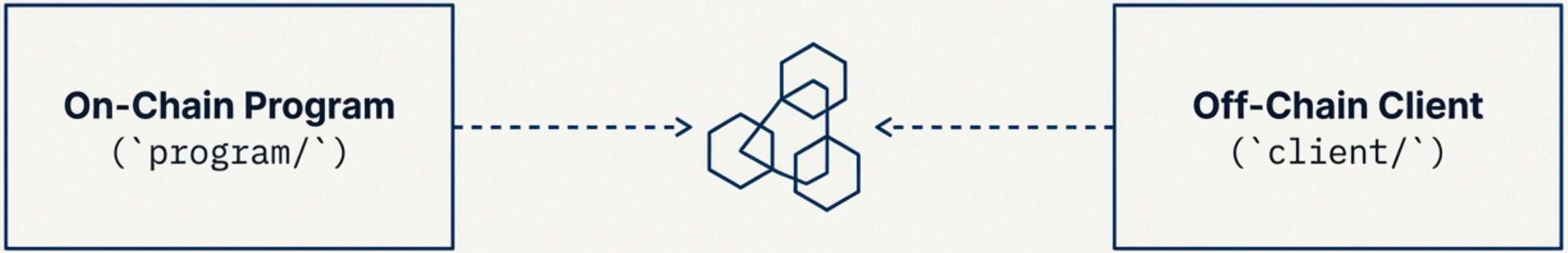
Decoding Class 7: From Code to a Live Program

A Step-by-Step Guide to Building, Deploying, and Interacting with Your First Solana Program



Your Mission: Master the Full Development Lifecycle

This guide decodes the lessons from our latest session (7B), building on the homework from 7A. The goal is to prepare you to present your learnings and demonstrate a complete understanding of the end-to-end workflow.

Key Objective
Successfully deploy the on-chain program to a `local validator`, use the `off-chain client` to call a function, and confidently explain every step in between.

The Two Halves of a Solana Application

The On-Chain Program

Directory: `program/`

What it is: The core logic that lives and executes on the Solana blockchain.

Analogy: The 'smart contract' or the 'backend server." It's immutable once deployed.

The Off-Chain Client

Directory: `client/`

What it is: The tool used to communicate with the on-chain program from a user's computer.

Analogy: The 'frontend' or 'remote control." It builds and sends transactions to the program.

Part 1: The Blueprint - Understanding the Workspace

The project is structured as a Rust ‘workspace,’ which allows us to manage our on-chain and off-chain code as a single, unified project. This is defined in the root `Cargo.toml` file.

```
[workspace]
members = [
    "program",
    "client"
]
```

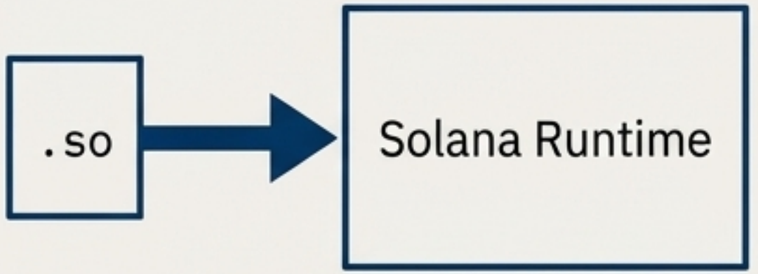
Mentor's Memo

Why a Workspace? It tells Rust that `program` and `client` are two separate packages (crates) that belong together. This simplifies building and managing dependencies for the entire application.

Library vs. Executable: Why `lib.rs` and `main.rs` are Different

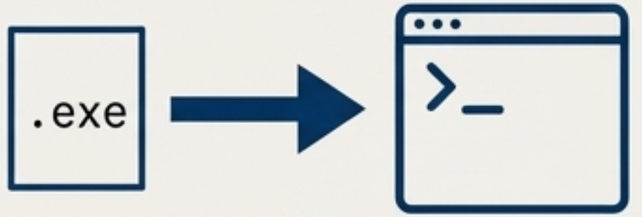
The Program is a Library (`lib.rs`)

- It's compiled as a **shared library** (`.so` file) for the Solana runtime to load and execute.
- It has no `main()` function. The `entrypoint!` macro declares the starting point for the runtime.
- The `program/Cargo.toml` specifies this with: `crate-type = ["cdylib"]`.



The Client is an Executable (`main.rs`)

- It's compiled as a standard **binary executable** that runs on your computer.
- It has a `fn main()` where its execution begins.
- Its purpose is to be run from your terminal to interact with the on-chain program.



Mentor's Memo

A program is a library *for* the Solana runtime. A client is an executable *for* your machine. This separation is fundamental.

Part 2: The Core Logic - Inside the On-Chain Program

The `program/src/lib.rs` file contains the program's entrypoint. The `process_instruction` function acts as a router, deciding what to do based on the data sent by the client.

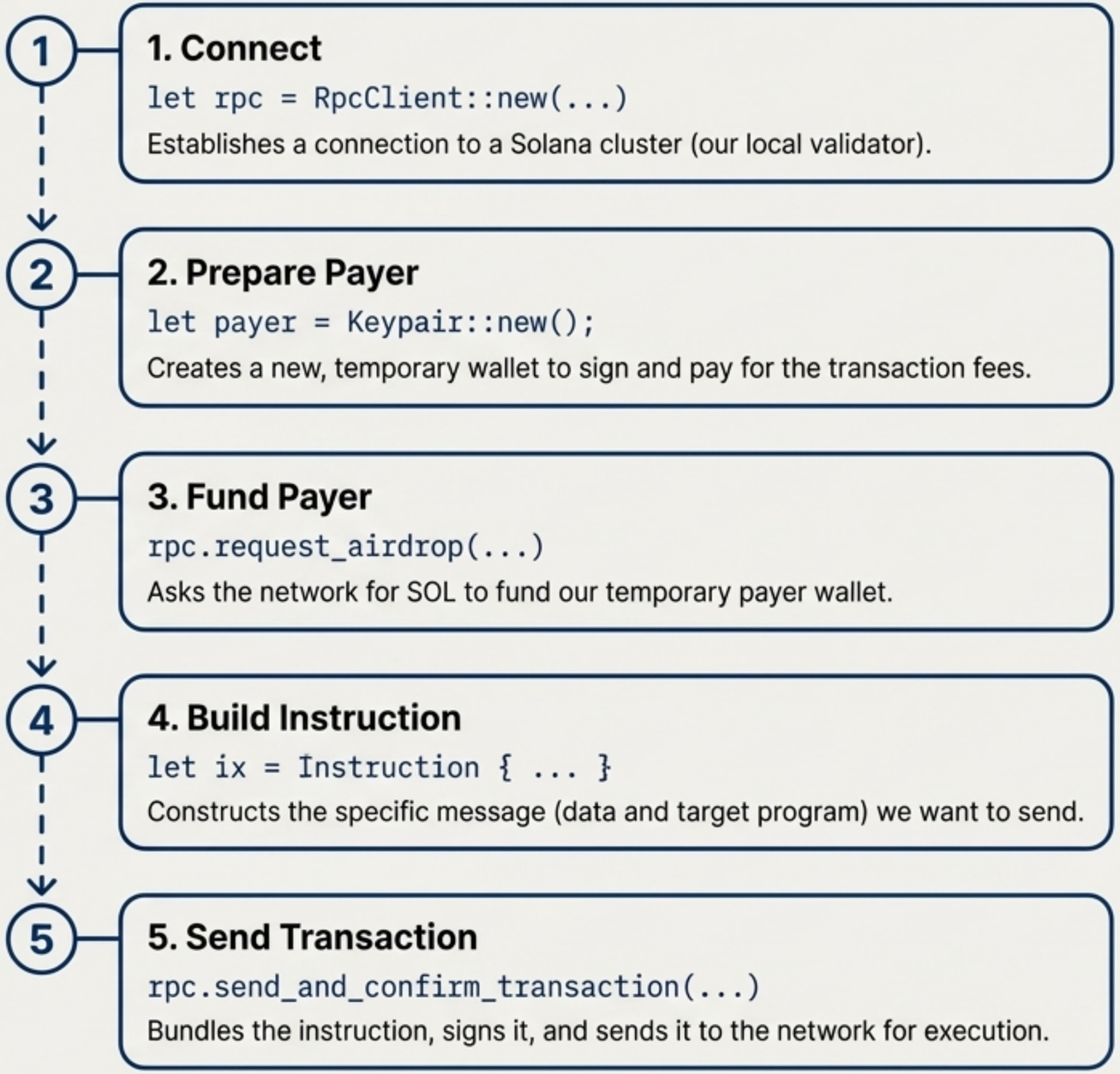
```
entrypoint!(process_instruction); // 1.

pub fn process_instruction(
    _program_id: &Pubkey,
    _accounts: &[AccountInfo],
    instruction_data: &[u8], // 2.
) -> ProgramResult {
    match instruction_data[0] { // 3.
        0 => deposit(...),  // 4.
        1 => withdraw(...), // 5.
        _ => Err(ProgramError::InvalidInstructionData)
    }
}
```

1. **Entrypoint**: Declares this function as the program's starting point.
2. **Instruction Data**: The raw byte array sent from our off-chain client.
3. **The Router**: We use the first byte of the data (`instruction_data[0]`) to determine which logic to execute.
4. **Route to `deposit`**: A '0' byte triggers the `deposit` function.
5. **Route to `withdraw`**: A '1' byte triggers the `withdraw` function.

Part 3: The Remote Control - Anatomy of the Client Script

The `client/src/main.rs` script executes a sequence of actions to communicate with our on-chain program.



The Critical Lesson: Solving the Airdrop Race Condition

The Problem

The error we saw yesterday was a race condition. The client script was faster than the network. It tried to send the `deposit` transaction *before* the airdrop transaction was finalized.

The payer wallet had 0 SOL and couldn't pay the gas fee, causing the transaction to fail.

The Fix: Wait for Finalization

Before (Problematic)

```
rpc.confirm_transaction(&airdrop_sig)
```

After (Correct)

```
// Fully wait for actual finalization
rpc.poll_for_signature(&airdrop_sig)
```

The Explanation

`confirm_transaction` only waits for a node to process the transaction. `poll_for_signature` waits for it to be fully finalized by the cluster. We must wait for the airdrop to be final before we can use the funds.

Mentor's Memo

Your code will always be faster than a distributed network. Always wait for prerequisite transactions (like an airdrop) to be fully finalized before proceeding.

Dissecting the Instruction

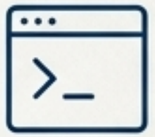
The `Instruction` struct is the core message sent to the Solana runtime. It packages everything the network needs to route and process our request.

```
let ix = Instruction {
    program_id,
    accounts: vec![],
    data: vec![0, 1, 1, 1],
};
```

Breakdown

- **`program_id`**: Tells the Solana network which specific on-chain program to send this instruction to.
- **`accounts: vec![]`**: A list of accounts the program needs to read or write. It's empty because our current `deposit` function doesn't interact with any accounts yet.
- **`data: vec![0, 1, 1, 1]`**: The payload for our program. The `0` routes to the `deposit` function inside our program. The `[1, 1, 1]` are placeholder bytes for the deposit amount we will add in a future lesson.

Part 4: The Launch Sequence - The Full Deployment Playbook



1. Configure CLI

`solana config set --url localhost`

Point your Solana CLI tool to your local test network.

2. Start Validator

`solana-test-validator`

Purpose: Run a local Solana cluster on your machine. (Keep this terminal running).

3. Build Program

Commands: `cd program` then `cargo build-sbf`

Purpose: Compile the on-chain Rust code into a deployable `.so` file.

4. Deploy Program

`solana program deploy target/deploy/luke_vault.so`

Upload the compiled program to your local validator. ****Copy the output Program ID!****

5. Update Client

Paste the new Program ID into the `program_id` variable in `client/src/main.rs`.

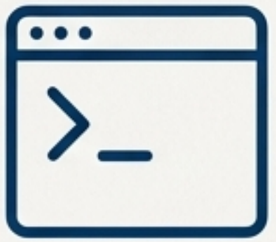
6. Run Client

Commands: `cd client` then `cargo run`

Purpose: Execute the client script to send a live transaction to your deployed program.

A Tale of Two Connections

It’s important to understand that the Solana CLI tool and our standalone Rust client connect to the network independently.



The Solana CLI

- **How it connects:** Uses the global configuration set by IBM `solana config set ...`.
- **Used for:** Commands run in your terminal, like `solana deploy` or `solana airdrop`.
- **To change network:** You must run the `solana config set` command again.

The Rust Client (IBM `main.rs`)

- **How it connects:** Uses the URL hardcoded inside the script: `RpcClient::new("http://127.0.0.1:8899")`.
- **Used for:** The program you run with `cargo run`.
- **To change network:** You must edit the URL string in the `.rs` file and recompile.

Mentor’s Memo

Don’t get confused. The CLI and your standalone script are two different tools with their own separate connections. Know which one you’re using and how it’s configured.

Your Assignment: Prepare Your Presentation

Be ready to teach this entire workflow back to me tomorrow. You will drive the screen share and explain each step.

****Preparation Checklist****:

- **Setup:** Ensure you have the latest code from the `study` branch of the repository.
- **Practice the Playbook:** Execute the full 6-step deployment sequence multiple times until you can do it smoothly.
- **Explain the Code:** Be ready to walk through `client/src/main.rs` line-by-line and explain the purpose of each code block.
- **Articulate the 'Why':** Clearly explain the airdrop race condition, why it happens, and how `poll_for_signature` is the correct solution.
- **Connect the Dots:** Describe exactly how the `data` vector in the client's instruction is received and used by the `match` statement in the on-chain program.

Beyond the Basics: What's Next on Our Journey

Mastering this workflow is the foundation. Once you've presented it, we will build upon it.

- **Tomorrow:** Your presentation and code walkthrough.
- **Next Session:** We will begin adding real logic to our on-chain program.
- **Future Goal:** Implement the IBM Plex `Monodeposit` and IBM Plex `withdraw` functions to actually handle user accounts and transfer IBM PlenoSOL, turning our simple program into a functional vault.