\*\*Scope Document: Self-Correcting AI-Driven Anchor Framework\*\*

### \*\*Objective\*\*

To develop an autonomous system that ensures:

1. Self-correction of errors during the installation and setup of the Anchor framework.

2. Continuous resolution of compilation errors in the `lib.rs` file during the build process.

3. A looped process that records, resolves, and retries compilation until all errors are resolved or the same error persists more than 5 times, triggering a failure.

---

### \*\*Key Requirements\*\*

#### \*\*1. Core Features\*\*

1. \*\*Installation Troubleshooting\*\*:

- Ensure all dependencies for Anchor are installed.

- If any errors occur during installation, the system:

- Parses error logs.

- Identifies the root cause.

- Automatically resolves the error (e.g., installs missing dependencies).

- Re-runs the installation process.

2. \*\*Compilation Error Handling for \\*\\*\\*\\*\\*\\*\\*\\*\*\*\*\*`lib.rs`\*\*:

- If `anchor build` fails, the system:

- Logs the error.

- Matches the error against a predefined error resolution database.

- Attempts to resolve the error by modifying the code or setup.

- Re-compiles until all errors are resolved or the same error occurs more than 5 times.

3. \*\*Kill List for Persistent Errors\*\*:

- Maintain a "kill list" for errors that:

- Cannot be resolved after 5 attempts.

- Logs these errors with detailed debugging information for manual intervention.

4. \*\*Autonomy\*\*:

- The framework operates autonomously, requiring minimal user input.

- Includes feedback loops to refine error resolution processes over time.

5. \*\*Offline Compatibility\*\*:

- Ensure the AI engine can operate offline with embedded error resolution logic and lightweight AI models.

6. \*\*Anchor Framework Customization\*\*:

- The entire system will be based on modifications and customizations of the Anchor framework source code to integrate AI-driven self-correction mechanisms.

---

### \*\*System Architecture\*\*

#### \*\*1. Components\*\*

1. \*\*Error Parser\*\*:

- Extracts relevant information from installation and compilation error logs.

- Uses regex and lightweight NLP for pattern matching and context understanding.

2. \*\*Error Resolution Engine\*\*:

- Maps errors to predefined solutions stored in a database.

- Uses AI to dynamically generate fixes for unknown errors.

3. \*\*Execution Layer\*\*:

- Executes corrective actions (e.g., re-running commands, modifying `lib.rs`, or installing dependencies).

4. \*\*Kill List Manager\*\*:

- Tracks unresolved errors and logs them for manual review after 5 failed attempts.

5. \*\*Feedback Loop\*\*:

- Updates the error resolution database based on unresolved errors and user feedback.

6. \*\*AI Engine\*\*:

- Lightweight NLP model to assist in error analysis and resolution.

- Embedded directly into the framework for offline functionality.

---

### \*\*Implementation Plan\*\*

#### \*\*1. Installation Troubleshooting\*\*

1. \*\*Error Detection\*\*:

- Parse logs from the Anchor installation process.

- Identify missing dependencies or configuration issues.

2. \*\*Error Resolution\*\*:

- Automatically resolve common errors, such as:

- Missing Rust installation.

- Missing Solana CLI setup.

- Version conflicts in dependencies.

- Example resolution steps:

- Install Rust: `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`

- Install Solana CLI: `sh -c "$(curl -sSfL https://release.solana.com/stable/install)"`

3. \*\*Retry Mechanism\*\*:

- Re-run the installation after applying fixes.

- Log all resolved and unresolved errors.

---

#### \\*\\*2. Compilation Error Handling for \\*\\*\*\*`lib.rs`\*\*

1. \*\*Error Parsing\*\*:

- Use regex or lightweight NLP to parse `cargo build` error logs.

- Extract key information, such as missing imports, unresolved references, or syntax errors.

2. \*\*Error Database\*\*:

- Maintain a database of common errors and their resolutions, e.g.,

```json

{

"error\_patterns": [

{

"pattern": "cannot find value",

"solution": "Add missing variable declaration."

},

{

"pattern": "missing crate",

"solution": "Add the crate to Cargo.toml."

}

]

}

```

3. \*\*AI Resolution\*\*:

- Use an embedded AI model to dynamically generate fixes for unknown errors.

- Example:

- Input: `error[E0425]: cannot find value 'xyz' in this scope`

- AI Suggestion: "Add `let xyz = ...` before the reference."

4. \*\*Kill List Management\*\*:

- If the same error persists after 5 attempts, log it in a kill list:

```json

{

"persistent\_errors": [

{

"error\_message": "error[E0425]: cannot find value 'xyz'",

"attempts": 5,

"log": "Detailed error log here."

}

]

}

```

---

#### \*\*3. Loop Process\*\*

1. \*\*Execution Flow\*\*:

- Parse errors → Resolve errors → Retry compilation.

2. \*\*Retry Logic\*\*:

- Loop through error handling and compilation up to 5 times.

- If successful, terminate the loop.

- If persistent errors remain, log them and halt.

---

### \*\*Development Tools\*\*

#### \*\*1. Core Technologies\*\*

- \*\*Language\*\*: Rust (Anchor framework), Python (AI engine).

- \*\*AI Model\*\*: Lightweight NLP model (e.g., DistilBERT, ONNX runtime).

- \*\*Error Database\*\*: JSON or SQLite.

#### \*\*2. Libraries and Tools\*\*

- \*\*Regex for Error Parsing\*\*: Rust regex crate.

- \*\*AI Frameworks\*\*: Hugging Face Transformers or ONNX.

- \*\*Execution Layer\*\*: Rust std::process for command execution.

#### \*\*3. Testing Tools\*\*

- Simulated environments for installation and compilation errors.

- Logging utilities for tracking error handling progress.

---

### \*\*Testing Plan\*\*

#### \*\*1. Installation Testing\*\*

- Simulate missing dependencies and misconfiguration.

- Verify that the system:

- Detects and resolves issues.

- Successfully completes the installation.

#### \*\*2. Compilation Testing\*\*

- Introduce syntax and dependency errors in `lib.rs`.

- Verify that the system:

- Parses and resolves errors.

- Logs unresolved errors after 5 attempts.

#### \*\*3. Stress Testing\*\*

- Test the framework with multiple concurrent installations and compilations.

- Ensure no crashes or performance degradation.

#### \*\*4. Offline Functionality\*\*

- Disable internet access and verify that the AI engine operates using local resources.

---

### \*\*Deliverables\*\*

1. Autonomous Anchor installation and setup system.

2. Self-correcting compilation process for `lib.rs`.

3. Error database with common patterns and solutions.

4. Embedded AI engine for dynamic error resolution.

5. Detailed logs for unresolved errors (kill list).

6. Documentation for usage, troubleshooting, and extending the system.

---

### \*\*Timeline\*\*

1. Week 1: Research and planning.

2. Week 2-3: Develop error parser and resolution engine.

3. Week 4: Integrate AI model for dynamic resolution.

4. Week 5: Implement and test loop process.

5. Week 6: Package and document the framework.

---

### \*\*Final Note\*\*

This system is designed to be autonomous, lightweight, and extensible. Future iterations can refine AI capabilities and expand the error database for improved performance.