# **Detailed Explanation of the Error-Aware SABRE Project**

This document explains precisely **what** is being done to enhance SABRE with hardware error awareness, and **how** each piece fits together.

## **1. Motivation & Objective**

* **SABRE** is a qubit-mapping and routing heuristic for quantum circuits. Traditionally, it only accounts for device connectivity, ignoring specific hardware error rates.
* The **goal** is to enrich SABRE so that both CNOT error rates and readout errors are incorporated into the mapping decision. That way, the transpiler can prefer qubits and couplings that have lower hardware error rates.

## **2. Overall Flow**

1. **Extract Error Rates** in Python:  
   * In sabre\_layout.py, we call a method \_get\_error\_map() which pulls data from a real backend if available, or uses dummy data for a fake backend.
   * The dictionary error\_map looks like {(q1,q2): cnot\_error\_rate, q: readout\_error}.
2. **Construct a Rust NeighborTable**:  
   * We pass error\_map into the NeighborTable constructor in neighbor\_table.rs.
   * Within Rust, we store edges, gate-error weights (via -3ln(1 - p)), readout errors, and run Floyd–Warshall to compute the full-pairwise distance matrix.
3. **Integrate with the SABRE Layout**:  
   * In layout.rs, we call sabre\_layout\_and\_routing(...), which receives the NeighborTable (already holding error data) and a DAG representation of the quantum circuit.
   * The function attempts multiple layout trials with different seeds and picks the best.
4. **Heuristic Adjustments**:  
   * In heuristic.rs, we define a method evaluate\_readout\_penalty(...), retrieving readout errors for the two qubits in a prospective swap.
   * The penalty is combined with standard SABRE distance scores.
5. **Swap Scoring & Selection**:  
   * In route.rs, the main loop references evaluate\_readout\_penalty(...) when evaluating each possible swap.
   * The best swap is chosen by combining connectivity/distance with the readout penalty.
6. **Transpiler Output**:  
   * Finally, a layout and swap map is returned to Python. The circuit is mapped to hardware with an improved chance of lower error.

## **3. Detailed Steps & Implementation Highlights**

### **3.1. sabre\_layout.py**

* **Extracting Error Data**: \_get\_error\_map(backend) determines if the backend is real or fake.

**Passing to Rust**:  
  
 neighbor\_table = NeighborTable(

rx.adjacency\_matrix(coupling\_map.graph),

error\_map=error\_map

)

**Heuristic & Layout**:  
  
 (initial\_layout, final\_permutation, sabre\_result) = sabre\_layout\_and\_routing(

sabre\_dag,

neighbor\_table,

dist\_matrix,

heuristic,

...

error\_map,

)

### **3.2. neighbor\_table.rs**

* **Parsing the error\_map**:  
  + For pairs (q1, q2), we interpret them as 2Q gate error → store in weights array after transformation -3 ln(1-error), which helps treat error as a distance-like quantity.
  + For single qubit keys q, interpret them as readout errors → store in readout\_errors vector.
* **Floyd–Warshall**: Fill out the distances array so we have short-paths across the hardware.
* **Methods** like get\_error\_distance() and get\_readout\_error() let the rest of the code query the stored data.

### **3.3. heuristic.rs**

* **evaluate\_readout\_penalty(...)**:  
  + Compute sum of readout errors for the two qubits in a prospective swap.
  + Multiply by a readout penalty weight, typically 1.0 for simplicity.

### **3.4. route.rs**

* **Core SABRE Loop**:  
  + choose\_best\_swap() iterates over candidate edges.
  + Adds up basic distance-based score + lookahead + readout penalty.
  + The final best swap is chosen by comparing total scores.
* **Main**: Distances from distance\_matrix, readout error from neighbor\_table, feed into heuristic.

### **3.5. layout.rs**

* **sabre\_layout\_and\_routing** orchestrates partial layout trials.
* Merges error-based weighting with random seeds, picking an optimal partial layout or final layout.

## **4. Debug Verification**

1. **Python Debug**:  
   * \_inner\_run prints the final error\_map before calling Rust.
2. **neighbor\_table.rs** logs each (key, value) from Python.
3. **heuristic.rs** prints readout penalty calculations.
4. **route.rs** logs final swap scores after adding readout penalty.

Everything shows consistent transformation from (0,1): 0.01 → 0.03015 weight, readout errors ~ 0.005, etc.

## **5. Conclusion**

* **What**: Extended SABRE to account for gate & readout errors.
* **How**: By injecting error map data from Python to Rust, storing them in NeighborTable, and factoring them into the heuristic.
* The final result is a transpiler pass that systematically chooses a layout and routing with reduced error risk.