NEXAR DEVELOPMENT PROGRESS REPORT

AI Code Converter

Project ID: 25-26J-484

Submitted Date: 17.12.2025

Submited By: Hettiarachchi S.R (IT22128386)

TABLE OF CONTENTS

PROJECT CONTEXT

The AI Code Converter serves as the intelligent translation layer within the NEXAR platform (Quantum-Classical Code Router). Its purpose is to automatically transform classical Python

code into equivalent Qiskit quantum implementations, enabling developers without quantum expertise to leverage quantum computing capabilities and see the execution comparisons. This component bridges the gap between classical and quantum programming paradigms through machine learning-based code translation.

MAIN OBJECTIVE

To develop an AI-powered code converter that automatically transforms classical Python logic gate implementations into functionally equivalent Qiskit quantum circuits, achieving 90%+ translation accuracy for target algorithm patterns while maintaining syntactic correctness and semantic equivalence.

WORK COMPLETED

1. MODEL SELECTION & ARCHITECTURE

Pre-trained Model Evaluation

After comprehensive analysis of available transformer-based code generation models, **CodeT5-small** was selected as the foundation model due to:

- **Superior Code Understanding**: CodeT5's encoder-decoder architecture with identifier-aware attention provides better code structure comprehension compared to decoder-only models
- **Training Stability**: Demonstrated consistent convergence during fine-tuning with lower variance in validation loss
- **Domain Adaptability**: Pre-trained on diverse code repositories, making it well-suited for learning quantum programming patterns
- **Resource Efficiency**: Small model size (220M parameters) provides optimal balance between performance and training feasibility
- **Production Readiness**: Robust inference performance suitable for API deployment

Model Architecture

- **Base Model**: CodeT5-small (Salesforce/codet5-small)
- **Fine-tuning Strategy**: Full model fine-tuning on curated quantum translation pairs
- **Token Handling**: Custom tokenizer configuration to support Qiskit-specific syntax
- **Training Framework**: PyTorch with Hugging Face Transformers

2. API IMPLEMENTATION

A robust microservice has been built using the FastAPI framework to handle translation and execution requests.

2.1. Core API Features:

- **Asynchronous Processing**: Non-blocking request handling for improved throughput
- **CORS Middleware**: Configured for seamless frontend integration
- **Comprehensive Error Handling**: Structured exception management with informative error responses
- **Validation Layer**: Pydantic schemas for request/response validation

2.2. Implemented Endpoints:

1. **POST /translate**
   - **Purpose**: Converts classical Python code to Qiskit quantum circuit
   - **Input**: JSON payload containing Python function code
   - **Output**: Generated quantum circuit code with metadata
   - **Processing**: CodeT5 model inference with post-processing validation
   - **Response Time**: ~200-500ms per translation

2. **POST /execute**
   - **Purpose**: Executes generated quantum circuit on Qiskit Aer simulator
   - **Input**: Quantum circuit code or translation result
   - **Output**: Execution results including measurement counts, probabilities, and circuit analysis
   - **Features**:
   - Configurable shot count (default: 1000)
   - Circuit depth and gate count analysis
   - Performance metrics comparison (Tells which one is executed faster)
   - Visualization data (circuit diagram, histogram)

3. **GET /health** (For Testing Purposes)
   - Service health check and model status verification

2.3 API Integration Architecture(Work Flow)

User Input (Python Code)
↓

POST /translate
↓
CodeT5 Model Inference
↓
Quantum Code Generation
↓
POST /execute
↓
Qiskit Aer Simulation
↓
Results + Performance Metrics

`

3. ADVANCED DATA PIPELINE & FEATURE ENGINEERING

3.1 DATA GENERATION PIPELINE

Data Scarcity Challenge

Addressing the lack of public classical-to-quantum translation datasets, a custom Physics-Informed Data Generation System was developed.

Generation Strategy

Base Template Approach

- Created 20 foundational classical-quantum code pair templates
- Templates cover 4 primary logic gate operations: **XOR, NOR, NOT, OR**
- Each template represents correct quantum implementation patterns

Augmentation Methodology

Expansion Factor: Generated 600 training pairs from 20 base templates (30x multiplication)

Augmentation Techniques:

- **Variable Renaming**: Systematic variation of variable identifiers while preserving logic
- **Code Formatting**: Different whitespace and indentation patterns

Quality Assurance

- Each generated pair validated for functional equivalence
- Quantum circuits tested on simulator to verify correct behavior
- Duplicate detection to ensure dataset diversity

Dataset Characteristics

- **Total Samples**: 600 classical-quantum code pairs
- **Coverage**: 4 logic gate types (XOR, NOR, NOT, OR)

- **Split Ratio**: 80% training / 20% validation (for the 2nd fine tuning used 60 / 40 split)
- **Format**: JSON structured pairs with metadata

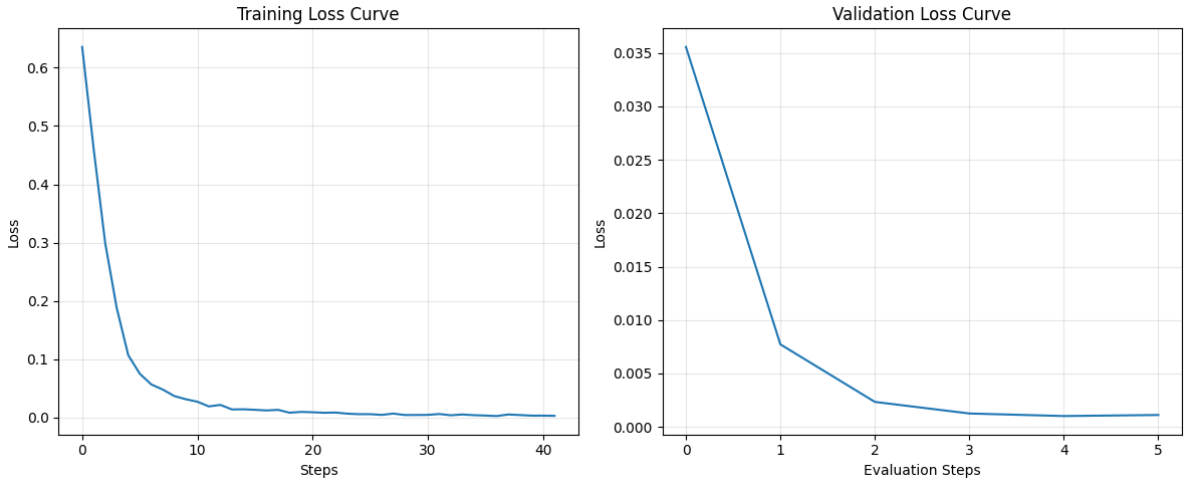
3.2 Modal Training & Performance

Training Configuration

- **Optimizer**: AdamW with weight decay (0.01)
- **Learning Rate**: 5e-5 with linear warmup and decay
- **Batch Size**: 8 (with gradient accumulation)
- **Epochs**: 40 total training epochs (20 each)
- **Hardware**: Training conducted on available GPU resources
- **Framework**: PyTorch with Hugging Face Transformers

Performance Metrics

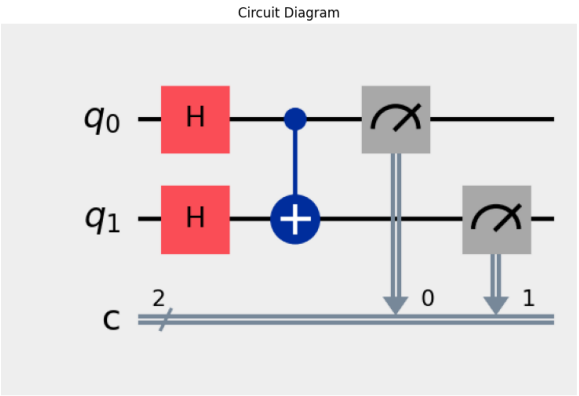
Training Results

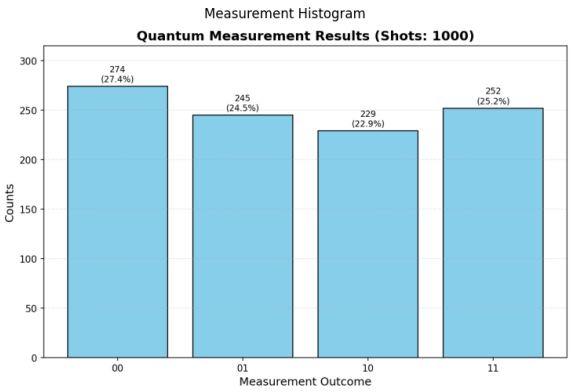


Key Observations - No divergence between training and validation curves (indicates healthy learning) . Validation loss decreased smoothly without erratic fluctuations

.

4.TEST SAMPLE RESULTS






```
{
  "test_metadata": {
    "gate_type": "OR",
    "test_timestamp": "2025-12-17 12:39:12",
    "shots_executed": 1000,
    "server_url": "http://127.0.0.1:8000"
  },
  "original_python_code": "def or_gate(a, b): return a | b",
  "translation_results": {
    "raw_quantum_code": "qc = QuantumCircuit(2, 2)\nqc.h(0)\nqc.h(1)\nqc.x(0)\nqc.cx(0,1)\nqc.measure([0,1],[0,1])",
    "quantum_code_with_imports": "from qiskit import QuantumCircuit\nqc = QuantumCircuit(2, 2)\nqc.h(0)\nqc.h(1)\nqc.x(0)\nqc.cx(0,1)\nqc.measure([0,1],[0,1])",
    "translation_success": true
  },
  "quantum_execution": {
    "success": true,
    "used_generated_code": true,
    "fallback_reason": null,
    "counts": {
      "11": 266,
      "00": 250,
      "10": 256,
      "01": 228
    },
    "probabilities": {
      "11": 0.266,
      "00": 0.25,
      "10": 0.256,
      "01": 0.228
    },
    "circuit_analysis": {
      "depth": 4,
      "num_qubits": 2,
      "num_gates": 6,
      "gate_counts": {
        "h": 2,
        "measure": 2,
```


5. CURRENT SYSTEM CAPABILITIES

Supported Translation Patterns

1. **XOR Gate**: Classical bitwise XOR → Quantum CNOT-based implementation
2. **NOR Gate**: Classical NOR logic → Quantum circuit with X and controlled operations
3. **NOT Gate**: Classical negation → Quantum X-gate
4. **OR Gate**: Classical bitwise OR → Quantum superposition-based implementation

System Strengths

1. **Robust API Infrastructure**: Production-ready FastAPI microservice
2. **End-to-End Pipeline**: Seamless translation-to-execution workflow

3.Performance Monitoring: Comprehensive metrics collection for both quantum and classical execution
4.Visualization Support: Automatic generation of circuit diagrams and result histograms

Current Limitations
1.Pattern Scope: Limited to 4 basic logic gate types
2.Semantic Validation: Translation quality assessment relies on syntactic correctness; functional equivalence validation is manual
3.Scalability: Model trained on relatively small dataset (600 pairs)
4.Complex Algorithms: Not yet tested on multi-gate compositions or advanced quantum patterns

6. PROJECT STRUCTURE & TECHNICAL INFRASTRUCTURE

The project follows a modular production-grade structure:

- `config/`: `app_config.py` with Pydantic settings for environment management.
- `routes/`: FastAPI route handlers separating the API logic from business logic.
- `services/`: Core business logic (Code Converter, Code validator, Code execution).
- `ml_models/`: CodeT5 small stored as Codet5-quantum
- `scripts/`: Testing scripts, Dataset Validation scripts, Performance Accuracy scripts

Environment: A virtual environment (`venv`) manages all dependencies, including **FastAPI/Uvicorn** for the web framework, and **Qiskit Aer simulator** for the simulation pipeline.

7. CHALLENGES FACED SO FAR

1.Lack of Available Data
There are very few publicly available datasets that convert classical Python code into Qiskit quantum code. Because of this, a custom dataset had to be manually created, which took extra time and effort.

2.Difference Between Classical and Quantum Logic
Classical programming logic is very different from quantum programming. Converting classical logic gates into correct quantum circuits while keeping the same behavior was challenging.

3.Small Training Dataset
The model was trained using a limited number of samples (600 code pairs). This makes it harder for the model to learn more patterns and generalize well to new inputs.

4.Difficulty in Semantic Validation
Automatically checking whether the generated quantum code behaves exactly the same as the classical code is difficult. Most validation is done using simulation results, and some manual checking is still required.

5.Limited Hardware Resources
Model training and fine-tuning were done with limited GPU resources. This restricted the use of larger models and required careful tuning of training parameters.

6.Handling Qiskit-Specific Syntax
Qiskit uses specialized syntax and functions. Supporting this syntax required additional tokenizer customization and post-processing to ensure valid quantum circuit generation.

7.Limited Support for Complex Logic
The current system supports only basic logic gates. Handling multiple gates together or more complex quantum algorithms is still a challenge and will be addressed in future work.

HM – Had to learn python from the very beginning