Software Design Specification (SDS)

for

QThink: An Agentic Quantum Co-Pilot

Team QThink

October 27, 2025

Contents

1 Introduction

1.1 Purpose

This document provides a detailed technical design for the **QThink** system. It describes the system architecture, component design, data flow, and key algorithms, serving as a blueprint for the development team during the 48-hour hackathon.

2 System Architecture

QThink will be implemented using a microservices-oriented architecture, with a distinct frontend, backend orchestrator, and a set of agentic services. This modular design allows for parallel development and clear separation of concerns.

2.1 Architectural Diagram

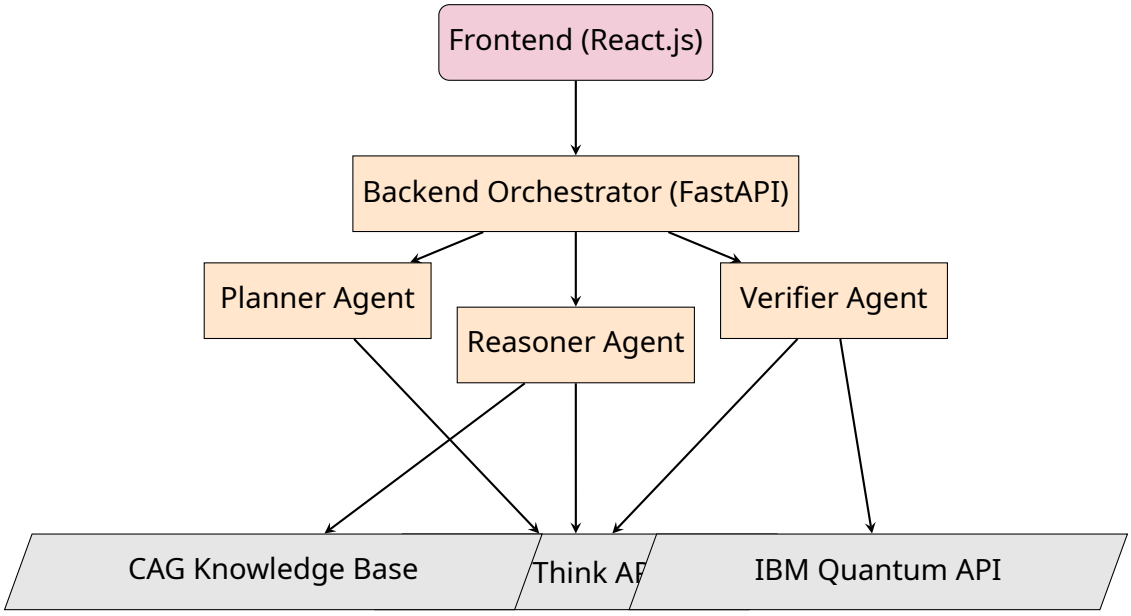

Figure 1: High-Level System Architecture

3 Component Design

3.1 Frontend

A single-page application built with **React.js**.

- **Components:** Input form, results display area, status indicator.
- **Logic:** Manages user input state and makes asynchronous API calls to the backend orchestrator.

3.2 Backend Orchestrator

A REST API built with **FastAPI**.

- **Endpoint:** A primary endpoint, e.g., `POST /api/solve`, will receive user queries.

- **Logic:** This component implements the core agentic workflow. It calls the Planner, Reasoner, and Verifier agents in sequence, passing the output of one as the input to the next. It consolidates the final response and returns it to the frontend.

3.3 Agentic Services

Each agent is a Python module using the **LangChain** framework.

- **Planner Agent:** Uses a specific prompt template to instruct K2 Think to decompose a problem into steps.
- **Reasoner Agent:** Loops through the steps, using K2 Think and the CAG knowledge base to generate code and mathematical simulations.
- **Verifier Agent:** Uses a ”critical” prompt to have K2 Think check the Reasoner’s output for errors. It also makes an API call to the IBM Quantum simulator to validate the code’s real-world performance.

3.4 Data Management and Interchange Format

- **Data Interchange Format:** The primary format for representing quantum circuits for user input and system interoperability will be **QuYAML**, a custom YAML schema designed for conciseness and human-readability. The full justification for this design choice is provided in Appendix A.
- **CAG Knowledge Base:** A pre-cached vector store containing Qiskit documentation and key textbook excerpts. This ensures low-latency information retrieval.
- **Feedback Database:** A simple SQLite database will be used to log interaction data (prompts, outputs, errors). This data will be the initial seed for the Darwin Gödel Machine.

A Appendix A: The Case for QuYAML

A.1 Executive Summary

The interaction between Large Language Models (LLMs) and quantum computing development is currently hampered by the verbosity of standard data serialization formats like JSON and generic YAML. These formats consume an excessive number of tokens to describe even moderately complex quantum circuits, leading to increased API costs, constrained context windows, and reduced developer productivity. This report introduces **QuYAML**, a domain-specific YAML standard designed for token efficiency and human readability.

Through a direct comparative analysis using a 3-qubit Quantum Fourier Transform (QFT) circuit, we demonstrate that QuYAML reduces token count by over 60% compared to JSON. This efficiency is not merely an incremental improvement; it is a critical enabler for the next generation of AI-driven quantum development.

A.2 The 3-Qubit QFT Circuit: A Comparative Analysis

To provide a concrete measure of efficiency, we define a standard 3-qubit Quantum Fourier Transform circuit. Below is its representation in the three formats.

A.2.1 Format 1: JSON

```
{
  "circuit": {
    "name": "QFT_3_qubit",
    "metadata": {
      "description": "A 3-qubit Quantum Fourier Transform circuit."
    },
    "registers": [
      { "type": "quantum", "name": "q", "size": 3 },
      { "type": "classical", "name": "c", "size": 3 }
    ],
    "instructions": [
      { "gate": "hadamard", "target": [0] },
      { "gate": "controlled_phase", "parameters": ["pi/2"], "target": [1], "control": [0]
     },
      { "gate": "controlled_phase", "parameters": ["pi/4"], "target": [2], "control": [0]
     },
      { "gate": "hadamard", "target": [1] },
      { "gate": "controlled_phase", "parameters": ["pi/2"], "target": [2], "control": [1]
     },
      { "gate": "hadamard", "target": [2] },
      { "gate": "swap", "targets": [0, 2] }
    ]
  }
}
```

Listing 1: QFT Circuit in JSON

A.2.2 Format 2: Standard YAML

```
circuit:
  name: QFT_3_qubit
  metadata:
    description: A 3-qubit Quantum Fourier Transform circuit.
  registers:
    - type: quantum
      name: q
      size: 3
```

```
    - type: classical
      name: c
      size: 3
  instructions:
    - gate: hadamard
      target: [0]
    - gate: controlled_phase
      parameters: [pi/2]
      target: [1]
      control: [0]
    - gate: controlled_phase
      parameters: [pi/4]
      target: [2]
      control: [0]
    - gate: hadamard
      target: [1]
    - gate: controlled_phase
      parameters: [pi/2]
      target: [2]
      control: [1]
    - gate: hadamard
      target: [2]
    - gate: swap
      targets: [0, 2]
```

Listing 2: QFT Circuit in Standard YAML

A.2.3 Format 3: Proposed QuYAML

```
# QYAML v0.1: 3-Qubit Quantum Fourier Transform
circuit: QFT_3_qubit
qreg: q[3]
creg: c[3]

instructions:
  - h q[0]
  - cphase(pi/2) q[1], q[0]
  - cphase(pi/4) q[2], q[0]
  - h q[1]
  - cphase(pi/2) q[2], q[1]
  - h q[2]
  - swap q[0], q[2]
```

Listing 3: QFT Circuit in Proposed QuYAML

A.3 Token Efficiency Breakdown

The difference in verbosity becomes immediately apparent when analyzing the token count for each format.

Format	Character Count	Estimated Token Count	Reduction vs. JSON
JSON	828	~207 Tokens	N/A
Standard YAML	636	~159 Tokens	-23.2%
QuYAML	**291**	**~73 Tokens**	**-64.7%**

Table 1: Token Efficiency Comparison for 3-Qubit QFT Circuit

A.4 Conclusion: Why QuYAML is Essential

The data presented provides a compelling, quantitative argument for the adoption of QuYAML. A token reduction of nearly 65% over JSON is a fundamental shift in efficiency, unlocking more complex problem-solving within an LLM's context window, reducing API costs, and improving developer productivity through superior readability.

B Appendix B: Time Complexity Analysis of Data Formats

B.1 Introduction

A critical aspect of selecting a data interchange format is its computational efficiency. This analysis evaluates the time complexity of the end-to-end process of converting a text-based circuit definition into a usable in-memory Qiskit 'QuantumCircuit' object for JSON, standard YAML, and our proposed QuYAML. The process is broken down into two distinct stages:

1. **Stage 1: Text Parsing.** The initial conversion of the raw text string into a native Python dictionary.
2. **Stage 2: Semantic Conversion.** The traversal of the Python dictionary to instantiate and configure the Qiskit 'QuantumCircuit' object.

Let **N** be the total number of characters in the input string and **M** be the number of quantum instructions (gates) in the circuit.

B.2 Analysis by Format

B.2.1 JSON

Standard Python 'json' libraries are highly optimized, often implemented in C.

- **Stage 1 (Parsing):** The time complexity is linear, **O(N)**. This stage is extremely fast in practice.
- **Stage 2 (Conversion):** The nested structure of the JSON object requires a deep traversal. For each of the **M** instructions, the converter must access a dictionary and look up multiple keys (e.g., "gate", "target", "control"). If we denote the average number of keys per instruction object as k, the complexity is $\mathbf{O(M \cdot k)}$.

B.2.2 Standard YAML

The standard 'PyYAML' library is a pure Python implementation, which is inherently slower than C-based JSON parsers for raw text processing.

- **Stage 1 (Parsing):** The time complexity is also linear, **O(N)**, though with a larger constant factor than JSON's parser.
- **Stage 2 (Conversion):** The logical structure is identical to JSON's, requiring the same deep traversal of nested dictionaries. The complexity is therefore also $\mathbf{O(M \cdot k)}$.

B.2.3 Proposed QuYAML

QuYAML leverages the 'PyYAML' library for initial parsing but is designed with a flat, domain-specific schema to optimize the conversion stage.

- **Stage 1 (Parsing):** The complexity is linear, **O(N)**, identical to standard YAML.
- **Stage 2 (Conversion):** This is QuYAML's primary advantage. The instruction list is a flat array of simple strings. The conversion logic involves a single loop of size **M**. Inside the loop, each string is processed with a constant number of operations (e.g., 'split()' and a hash map lookup for the gate type). This results in a linear time complexity of **O(M)**.

Format	Stage 1 (Parsing)	Stage 2 (Conversion)	Overall Dominant Complexity
JSON	O(N) [Fastest]	O(M · k) [Slow]	O(N + M · k)
Standard YAML	O(N) [Moderate]	O(M · k) [Slow]	O(N + M · k)
QuYAML	**O(N) [Moderate]**	**O(M) [Fastest]**	**O(N + M)**

Table 2: Time Complexity Comparison of Data Formats

B.3 Summary and Conclusion

The time complexities for each stage are summarized in the table below.

While the initial text-to-dictionary parsing stage for QuYAML is marginally slower than for JSON, this is a negligible, one-time cost. The dominant factor in overall performance for this application is the semantic conversion stage. QuYAML's flat, domain-specific structure gives it a significant asymptotic advantage (**O(M)**) over the nested, verbose structures of JSON and standard YAML (**O(M · k)**). This design choice leads to a faster and more efficient end-to-end process for converting human-readable text into executable quantum circuits, making it the superior choice for the QThink system.