A Beginner's Guide to Quantum Programming with IBM Qiskit: From Setup to Full Workflow

Section 1: An Introduction to the Quantum World with Qiskit

The field of quantum computing represents a fundamental shift in computation, promising to solve certain problems that are intractable for even the most powerful classical supercomputers. Gaining access to this power requires a specialized toolkit that can translate classical programming concepts into the language of quantum mechanics. Qiskit (Quantum Information Software Kit) stands as the world's most popular open-source software stack for this very purpose, providing a comprehensive gateway for developers, researchers, and enthusiasts to program quantum computers.[1] This report serves as a definitive, beginner-friendly guide to navigating the Qiskit ecosystem, establishing a robust development workflow from initial setup to the execution of complete quantum programs on both simulators and real quantum hardware.

1.1 What is Qiskit? Demystifying the Ecosystem

At its core, Qiskit is a Python-based software development kit (SDK) originally developed by IBM Research and first released in 2017.[1] It is designed to be hardware-agnostic, allowing users to build and run quantum programs on a variety of quantum systems, including IBM's own superconducting qubit processors.[3] However, the term "Qiskit" refers to more than just the SDK; it encompasses a broad collection of software tools that form a complete ecosystem for quantum computation.[1] Understanding the key components of this stack is the first step toward effective quantum programming.

- **Qiskit SDK (qiskit)**: This is the foundational library and the primary interface for any Qiskit user. Distributed as a Python package, the SDK provides the tools to construct and manipulate the fundamental elements of quantum programs. This includes creating QuantumCircuit objects, which are the blueprints for quantum computations, defining quantum logic gates and operations, and working with quantum states and operators through its quantum information module.[1] The SDK is where developers spend most of their time designing their quantum algorithms.

- **Qiskit Aer (qiskit-aer)**: Before running a program on expensive and in-demand quantum hardware, it is essential to test, debug, and validate it. Qiskit Aer is the official high-performance simulation engine for Qiskit that serves this purpose.[1] It runs locally on a classical computer and provides various simulation backends. The most common,
  qasm_simulator, emulates the execution of a quantum circuit and returns measurement counts, just as a real device would.[6] Other simulators, like the statevector_simulator, can provide the complete mathematical description of the final quantum state, which is invaluable for debugging but impossible to obtain from real hardware.[6] Critically, Aer can also simulate the effects of noise, allowing developers to anticipate how their circuits might perform in a real-world, error-prone environment.[1]
- **Qiskit Runtime (qiskit-ibm-runtime)**: This component represents a significant evolution in how users interact with cloud-based quantum hardware. Early models of cloud quantum computing involved a user sending a single circuit from their local machine to a quantum data center, waiting in a queue, receiving the result, and then repeating the entire process for the next step of an algorithm.[3] This introduced immense latency, making iterative algorithms—which form the backbone of many useful quantum applications—prohibitively slow.
  Qiskit Runtime solves this problem by using a cloud computing concept known as "containerization".[3] Instead of sending individual circuits, the user packages their entire quantum program, including any classical pre- or post-processing, into a container that is sent to the cloud and executed in an environment physically close to the quantum processing unit (QPU). This drastically reduces the communication latency between classical and quantum resources, enabling more complex and efficient workflows.[1] The modern way to interact with Qiskit Runtime is through a set of streamlined interfaces called
  Primitives, which are optimized for the most common quantum tasks.[5]
- **Qiskit Serverless and the Broader Ecosystem**: Beyond these core components, the Qiskit ecosystem continues to expand. Qiskit Serverless, for instance, provides tools to orchestrate complex workloads across a mix of quantum and classical resources (QPUs, CPUs, and GPUs), paving the way for true quantum-centric supercomputing.[5] The ecosystem also includes a variety of community-developed and IBM-developed packages that add specialized

functionality, such as advanced error mitigation techniques.[1]

1.2 The Quantum Programming Workflow: A Mental Model

To provide a structured approach to building quantum programs, the Qiskit development team has introduced a conceptual framework known as **Qiskit Patterns**. This four-step workflow serves as a mental model that organizes the entire process of taking a problem and solving it on a quantum computer.[7] This report will be structured around this powerful and intuitive model.

1. **Map**: The first step is to translate the problem into a format that a quantum computer can understand. This almost always involves designing a QuantumCircuit. This circuit encodes the problem's inputs and specifies the sequence of quantum gates that constitute the algorithm.[7]
2. **Optimize**: A circuit designed in the "Map" phase is an abstract, idealized representation. Real quantum hardware has physical limitations, such as which qubits can interact with each other and which specific gate operations are natively supported.[1] The "Optimize" step uses a powerful tool called the **Transpiler** to rewrite the ideal circuit into an equivalent one that is optimized for execution on a specific target backend. This crucial step aims to reduce the circuit's depth and gate count to minimize the impact of noise.[5]
3. **Execute**: Once the circuit is optimized, it is sent to a backend for execution. This could be a local simulator from Qiskit Aer or a real quantum device accessed via Qiskit Runtime.[7] The execution is typically managed by one of the Qiskit Primitives, such as the
   Sampler (for obtaining measurement outcome distributions) or the Estimator (for calculating expectation values of observables).[5]
4. **Analyze**: The quantum computer returns classical data—measurement counts or expectation values. The final step is to post-process this data, analyze it, and interpret the results in the context of the original problem. This often involves statistical analysis and visualization, for which Qiskit provides helpful tools.[7]

A key point for any beginner navigating online tutorials is the evolution of the Qiskit API. Many older examples use a function called execute() and a provider called IBMQ. While this code may still function, the modern, standard, and most efficient workflow is built upon the Qiskit Runtime service and its Sampler and Estimator primitives.[5] This report will exclusively teach this modern, primitive-based approach, as it represents the best practice for performance and is the future direction of the platform, thereby providing the user with durable and forward-looking knowledge.

Section 2: Setting Up Your Quantum Development Environment

A smooth and successful journey into quantum programming begins with a correctly configured development environment. Installation issues and dependency conflicts are common hurdles that can discourage newcomers. This section provides a single, robust, step-by-step "golden path" for setting up a stable and reliable Qiskit environment, designed to eliminate these potential points of friction and ensure a seamless start.

2.1 System Prerequisites: The Foundation

Before beginning the installation, it is essential to ensure the host system meets the necessary requirements. Qiskit is a cross-platform toolkit but has specific dependencies on the Python version and operating system architecture.[14]

- **Python Version**: Qiskit requires a modern version of Python. As of the latest releases, Python 3.9 or newer is required.[11] It is critical to adhere to the supported versions listed on the official Qiskit PyPI (Python Package Index) page, as using an unsupported Python version (either too old or too new) is a frequent cause of installation failures.[14]
- **Operating System**: Qiskit is officially tested and supported on the following 64-bit operating systems:
    - Windows 7 or later
    - macOS 10.12.6 or later
    - Linux (specifically Ubuntu 16.04 or later, though other distributions are likely to work).[14]

2.2 The Recommended Path: Anaconda and Virtual Environments

While Qiskit can be installed in a system's global Python environment, this is strongly discouraged. The recommended and most robust approach is to use the Anaconda distribution in conjunction with virtual environments.

Anaconda is a free, cross-platform distribution of Python tailored for scientific computing.[16] It simplifies the management of complex packages and their dependencies and comes bundled with

Jupyter Notebook, an interactive, web-based coding environment that is ideal for the exploratory and visual nature of quantum programming.[14]

A **virtual environment** is an isolated directory that contains a specific version of Python and a dedicated set of installed packages. Using a virtual environment for your Qiskit projects is a critical best practice. It ensures that the specific versions of Qiskit and its many dependencies do not conflict with other Python projects on your system.[18] This isolation prevents the kind of version-mismatch errors that can otherwise be difficult to diagnose and resolve.[15]

The following steps outline the process for setting up this recommended environment:

1. **Download and Install Anaconda**: Navigate to the official Anaconda website and download the graphical installer for your operating system. Follow the installation prompts, accepting the default settings. This will install Python, the Anaconda Navigator, and the conda command-line tool.[14]
2. **Create a Dedicated Conda Environment**: Open the Anaconda Prompt (on Windows) or your standard terminal (on macOS/Linux). Create a new virtual environment specifically for Qiskit by running the following command. It is good practice to specify a Python version known to be compatible, for example, 3.10.
   Bash
   ```bash
   conda create -n qiskit_env python=3.10
   ```

   This command creates a new environment named qiskit_env.[18]
3. **Activate the Environment**: Before you can use the new environment, you must "activate" it. This command must be run in every new terminal session where you intend to work with Qiskit.
   Bash
   ```bash
   conda activate qiskit_env
   ```

   Once activated, your terminal prompt will typically be prefixed with (qiskit_env), indicating that any Python or pip commands will now operate within this isolated environment.[18]

2.3 Installing Qiskit with All the Trimmings

With the virtual environment activated, the next step is to install the Qiskit packages using pip, the Python package installer.

1. **Install the Core Qiskit Package**: The main Qiskit SDK can be installed with a simple command.[11]
   Bash
   ```bash
   pip install qiskit
   ```

2. **Install Visualization Support (Highly Recommended)**: To enable the rich visualization features of Qiskit, such as plotting circuit diagrams and result histograms, it is essential to install the visualization "extra." This bundles matplotlib and other necessary libraries.[14]
   Bash
   ```bash
   pip install 'qiskit[visualization]'
   ```

   Note for users of the Zsh shell (the default on modern macOS): the square brackets must be enclosed in single quotes to prevent the shell from interpreting them as a special character.[14]

2.4 Verification: Is Everything Working?

The final step is to verify that the environment is correctly configured and Qiskit is installed properly.

1. **Launch Jupyter Notebook**: From your activated qiskit_env terminal, launch the Jupyter Notebook interface.
   Bash
   ```bash
   jupyter notebook
   ```

   This will open a new tab in your web browser showing the Jupyter file navigator.[14]
2. **Create and Run a Test Notebook**: Create a new Python 3 notebook. In the first

cell, enter the following code and execute it (by pressing Shift+Enter).

```python
import qiskit
print(qiskit.__version__)
```

If the installation was successful, this code will execute without any errors and will print the installed version of the Qiskit package.[16] This "smoke test" confirms that your quantum development environment is ready for use.

To provide a quick and scannable reference, the essential commands for this setup process are summarized in the table below.

Action	Command	Notes
Create Conda Environment	conda create -n qiskit_env python=3.10	Creates an isolated environment named qiskit_env with a specific Python version.
Activate Environment	conda activate qiskit_env	Must be done in every new terminal session before using Qiskit.
Install Qiskit	pip install 'qiskit[visualization]'	Installs the core SDK and visualization libraries. Use quotes on macOS/Linux.
Launch Jupyter	jupyter notebook	Must be run from the activated environment.

Section 3: The Building Blocks of a Qiskit Program (The "Map" Step)

With a functional development environment, the journey into writing quantum code can begin. This section delves into the "Map" phase of the Qiskit Patterns workflow, introducing the fundamental objects and operations used to describe a quantum computation. These are the core components that translate an algorithmic idea into a concrete set of instructions for a quantum computer.

3.1 The QuantumCircuit: Your Canvas

The central object in any Qiskit program is the QuantumCircuit. It serves as the canvas or blueprint upon which a quantum algorithm is built. It is a container that holds all the components of the computation: the quantum and classical bits (registers) and the sequence of quantum operations (gates) that act upon them.[17]

A QuantumCircuit is initialized by specifying the number of quantum bits (qubits) and, optionally, the number of classical bits it will contain.

Python

```python
from qiskit import QuantumCircuit

# Initialize a circuit with 3 qubits and 3 classical bits
qc = QuantumCircuit(3, 3)
```

This simplified initialization is the most common and recommended way to start.[21] The qubits are used to perform the quantum operations, while the classical bits are typically used to store the results of measurements.

3.2 Qubits and Classical Bits: The Performers and the Scorecard

- **Qubits**: A qubit is the fundamental unit of quantum information, analogous to the classical bit.[9] However, unlike a classical bit which can only be in the state 0 or 1, a qubit can exist in a
  superposition of both states simultaneously. Mathematically, the state of a single qubit $|\psi\rangle$ is described as a linear combination of its basis states, $|0\rangle$ and $|1\rangle$: $|\psi\rangle=\alpha|0\rangle+\beta|1\rangle$, where α and β are complex numbers called probability amplitudes. When measured, the qubit will "collapse" to the state $|0\rangle$ with probability $|\alpha|2$

and to the state $|1\rangle$ with probability $|\beta|2$.[9] In a circuit diagram, qubits are represented as horizontal wires.

- **Classical Bits**: These are the standard bits of classical computing, holding a definite value of either 0 or 1. In the context of a QuantumCircuit, their primary role is to act as a scorecard, recording the definite outcome that results from measuring a qubit.[9] This transfer of information from the quantum realm (qubits) to the classical realm (classical bits) is a necessary step for retrieving the result of a computation.

3.3 Essential Quantum Gates: The Instructions

Quantum gates are the fundamental operations that manipulate the state of qubits. They are the building blocks of any quantum algorithm, analogous to logic gates (like AND, OR, NOT) in classical computing.[9] For a beginner, mastering a small but essential set of gates is sufficient to build interesting and powerful circuits.

- **The Hadamard Gate (.h())**: This is arguably the most important single-qubit gate. Its primary function is to create superposition. When a Hadamard gate (H-gate) is applied to a qubit that is initially in the $|0\rangle$ state, it transforms it into an equal superposition of $|0\rangle$ and $|1\rangle$. This means that upon measurement, the qubit has a 50% chance of being found in state 0 and a 50% chance of being found in state 1.[9] This ability to explore multiple possibilities at once is a key source of quantum computing's power.
- **The Pauli-X Gate (.x())**: The Pauli-X gate (or X-gate) is the quantum equivalent of the classical NOT gate. It flips the state of a qubit, transforming $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$.[9]
- **The CNOT Gate (.cx())**: The Controlled-NOT gate (or CNOT gate) is a fundamental two-qubit gate and is essential for creating the uniquely quantum phenomenon of **entanglement**. It operates on a *control* qubit and a *target* qubit. The logic is simple: if the control qubit is in the state $|1\rangle$, an X-gate (a flip) is applied to the target qubit. If the control qubit is in the state $|0\rangle$, nothing happens to the target qubit.[9] By applying a CNOT gate to a control qubit that is in a superposition, we can create a correlated, entangled state between the two qubits.

3.4 Measurement (.measure()): Reading the Result

After applying a sequence of gates, the final step in most algorithms is to extract a classical result. The measure operation achieves this. It is the process that forces a qubit to collapse out of its superposition and yield a definite classical value of either 0 or 1. This classical result is then stored in a specified classical bit.[9]

The syntax in Qiskit maps specific qubits to specific classical bits:

Python

```python
# Measure qubit 0 and store the result in classical bit 0
qc.measure(0, 0)

# Measure a range of qubits and store them in a corresponding range of classical bits
qc.measure(, )
```

This action is the irreversible bridge from the quantum state of the system to the classical information that we can read and analyze.[17]

The following table provides a quick reference for these fundamental gates.

Gate Name	Qiskit Method	Purpose
Hadamard	.h(qubit)	Creates a superposition of $
Pauli-X (NOT)	.x(qubit)	Flips the qubit's state ($
CNOT (Controlled-NOT)	.cx(control_qubit, target_qubit)	Flips the target qubit if the control qubit is $

Section 4: Your First Program: "Hello, Entanglement!" with a Bell State

Having covered the theoretical building blocks, it is time to assemble them into a complete quantum program. The quantum equivalent of a "Hello, World!" program is not printing text but creating a **Bell state**. This is the simplest and most famous example of a two-qubit entangled state, and building it provides a perfect first hands-on coding experience in Qiskit.[17]

4.1 Objective: Creating Quantum Entanglement

Quantum entanglement is a phenomenon where two or more qubits become linked in such a way that their fates are intertwined, regardless of the distance separating them. The Bell state is a specific entangled state of two qubits where their measurement outcomes are perfectly correlated. If the two qubits are prepared in a Bell state, they will either both be measured as 0, or both be measured as 1. The outcomes 01 and 10 are impossible. Each of the correlated outcomes, 00 and 11, occurs with a 50% probability.[17] This perfect correlation is the signature of entanglement.

The recipe to create this state is straightforward and uses the gates introduced in the previous section:

1. Start with two qubits, both in the $|0\rangle$ state.
2. Apply a Hadamard gate to the first qubit to put it into a superposition.
3. Apply a CNOT gate, using the first qubit as the control and the second as the target.

4.2 Line-by-Line Code Walkthrough

The following script implements the creation of a Bell state in Qiskit. Each line is commented to explain its purpose, connecting the code directly to the concepts.

Python

```
# 1. Import the necessary components from Qiskit.
# QuantumCircuit is the main object for building the program.
from qiskit import QuantumCircuit

# 2. Initialize the circuit.
# We need 2 qubits to entangle and 2 classical bits to store the measurement results.
qc = QuantumCircuit(2, 2)

# 3. Apply a Hadamard gate to the first qubit (indexed as 0).
# This puts qubit q_0 into an equal superposition of |0> and |1>.
qc.h(0)

# 4. Apply a CNOT gate to create entanglement.
# Qubit q_0 is the control and qubit q_1 is the target.
# If q_0 is |1>, q_1 will be flipped. This links the two qubits' states.
qc.cx(0, 1)

# 5. Measure both qubits.
# The measurement of qubit 0 is stored in classical bit 0.
# The measurement of qubit 1 is stored in classical bit 1.
# The list  maps qubits to classical bits in order.
qc.measure(, )
```

This short script contains all the quantum instructions needed to generate and measure an entangled state.[17]

4.3 Visualizing Your Creation: The Circuit Diagram

Before executing the program, it is invaluable to visualize it to ensure it has been constructed correctly. The draw() method of the QuantumCircuit object is an essential tool for debugging and communication.[17]

By default, draw() produces a simple text-based diagram.

Python

```
print(qc)
```

While functional, a much clearer, publication-quality image can be generated by specifying the mpl (matplotlib) output format. This requires the qiskit[visualization] package to be installed, as covered in the setup section.

```python
# Display the circuit diagram using matplotlib
qc.draw(output='mpl')
```

This command will produce a diagram where each qubit (q_0, q_1) and classical bit (c_0, c_1) is represented by a horizontal line. The gates (H for Hadamard, CX for CNOT) appear as boxes on the qubit wires in the order they were applied. The measurement operations are shown by a symbol connecting the qubit wire to its corresponding classical bit wire.[24] This visual representation confirms that the circuit correctly implements the steps for creating a Bell state.

Section 5: The Execution Workflow: Simulators and Results ("Execute" & "Analyze")

With the Bell state circuit constructed, the next phases of the workflow are "Execute" and "Analyze." This section covers how to run the circuit and interpret its results, beginning with the ideal, noiseless environment of a local simulator. This is the standard practice for testing and validating any quantum program before considering real hardware.

5.1 Understanding Backends: Where to Run Your Code

In Qiskit, a **Backend** is the entity that executes a quantum circuit. A backend can be either a classical software simulator or a physical quantum computer.[6] Access to these backends is managed by a

Provider. For the purposes of this guide, there are two main providers to consider:

- **Aer Provider**: This provider gives access to the high-performance simulators that are included with Qiskit and run locally on your machine. The primary backend here is the AerSimulator.[12]
- **IBM Quantum Provider (qiskit-ibm-runtime)**: This provider facilitates access to IBM's cloud-based quantum systems, which include both advanced simulators and real quantum hardware.[12]

The choice between a local simulator and real hardware depends entirely on the task at hand. The following table provides a clear comparison to guide this decision.

Attribute	Local Simulator (e.g., AerSimulator)	Real Quantum Device (e.g., ibm_brisbane)
Backend Type	Software running on a local computer	Physical hardware accessed via the cloud
Execution Speed	Very fast; results are nearly instantaneous	Slower; involves network latency and job queuing
Results Quality	Ideal and noiseless; perfectly follows the laws of quantum mechanics	Noisy; results are affected by gate errors, decoherence, and readout errors
Cost	Free	May incur costs under a pay-as-you-go plan
Primary Use Case	Algorithm design, testing, debugging, learning, and ideal-world validation	Quantum research, hardware benchmarking, and running problems that leverage real quantum effects

For beginners, and for the vast majority of development and testing, the local AerSimulator is the correct choice. It provides rapid feedback and perfect, predictable results that are essential for learning and verifying that an algorithm is logically correct.

5.2 Running on a Local Simulator with the Sampler Primitive

The modern and standardized way to execute circuits and obtain measurement outcomes is by using the **Sampler primitive**. The Sampler's job is to take one or more quantum circuits, run them a specified number of times (called "shots"), and return the probability distribution of the measurement outcomes.[11]

The following code demonstrates how to use the Sampler from the Qiskit Aer provider to run the Bell state circuit created in the previous section.

Python

```python
# Import the Sampler from the Aer provider
from qiskit_aer.primitives import Sampler

# 1. Create an instance of the local Sampler
sampler = Sampler()

# 2. Define the number of times to run the circuit
# More shots lead to better statistical accuracy
shots = 1024

# 3. Submit the job to the Sampler
# The run() method takes the circuit(s) and runtime options like 'shots'
job = sampler.run(qc, shots=shots)

# 4. Retrieve the results from the completed job
# The.result() method is a blocking call; it waits until the job is finished.
result = job.result()

# The result object contains the output data
print(result)
```

The job object represents the execution task submitted to the backend. Once the job is complete, the result object contains all the output data and associated metadata.[12]

5.3 Interpreting Your First Results: The Histogram

The Sampler result object stores the outcome distributions. For a single circuit, this data can be accessed and then visualized using the plot_histogram function, a key tool in the "Analyze" phase.[13]

Python

```python
# Import the visualization function
from qiskit.visualization import plot_histogram

# 1. Extract the probability distribution from the result object
# For a single circuit, this is the first (and only) distribution in the list.
# The.binary_probabilities() method returns a dictionary of outcomes and their frequencies.
counts = result.quasi_dists.binary_probabilities()
print(f"Counts: {counts}")

# 2. Plot the results as a histogram
plot_histogram(counts, title="Bell State Measurement Outcomes")
```

When this code is executed, it will first print a dictionary-like object showing the measured bitstrings as keys and their probabilities as values. For an ideal simulation of the Bell state, this will look something like {'00': 0.5, '11': 0.5}.

The plot_histogram function then generates a bar chart based on this data.[17] The x-axis lists the possible measurement outcomes (

00, 01, 10, 11), and the y-axis shows the probability of observing each outcome. For the Bell state, the histogram will show two bars of approximately equal height at 00 and 11, and effectively zero-height bars at 01 and 10. The small deviations from a perfect 50/50 split are due to the statistical nature of sampling; running with a higher number of shots would cause the probabilities to converge even closer to the theoretical 0.5.[17] This visual result provides clear, unambiguous confirmation that the circuit successfully created the entangled Bell state.

Section 6: Advancing to Real Quantum Hardware (The Full "Optimize, Execute, Analyze" Flow)

After successfully building and simulating a quantum program, the ultimate goal is to run it on real quantum hardware. This section guides the user through the complete workflow for executing the Bell state circuit on an IBM Quantum computer. This process introduces the critical "Optimize" step, managed by the transpiler, and provides a profound, practical lesson on the impact of noise in real-world quantum systems.

6.1 Connecting to the IBM Quantum Platform

To access IBM's quantum hardware, a user must first have an account on the IBM Quantum platform.

1. **Create an Account**: Navigate to the IBM Quantum website (quantum.ibm.com) and sign up for a free account. This provides access to open-plan systems and simulators.[16]
2. **Retrieve Your API Token**: Once logged in, navigate to the account settings page. Here, you will find a unique API token. This token is a secret credential that authorizes your Qiskit instance to send jobs to your account.[16]
3. **Authenticate in Qiskit**: The modern way to connect to the service is by using the QiskitRuntimeService class. The first time you connect, you should save your credentials to a local configuration file. This only needs to be done once per machine.

   Python
   ```python
   from qiskit_ibm_runtime import QiskitRuntimeService

   # Save your credentials for future use (replace with your actual token)
   # This line should only be run once.
   # QiskitRuntimeService.save_account(channel="ibm_quantum", token="YOUR_API_TOKEN_HERE")

   # In all subsequent sessions, you can simply initialize the service to load your saved credentials.
   service = QiskitRuntimeService(channel="ibm_quantum")
   ```

This service object is now your gateway to all the backends available through your IBM Quantum account.[28] This method replaces the older IBMQ.save_account() and IBMQ.load_account() functions seen in legacy tutorials.[19]

6.2 The Transpiler: Adapting Your Circuit for Reality (The "Optimize" Step)

The concept of transpilation is what separates theoretical quantum computing from practical application. An abstract circuit, like the one for the Bell state, makes two key assumptions: that any gate can be applied, and that any pair of qubits can interact. Real quantum chips do not have these luxuries. They have physical constraints that must be respected.[1]

- **Native Gate Set**: A physical QPU can only directly implement a small, specific set of quantum gates, known as its "native gate set." Any other gate in your circuit (like a Hadamard gate, if it's not native) must be broken down into an equivalent sequence of native gates.[29]
- **Connectivity (Coupling Map)**: On a quantum chip, qubits are physical entities. Not all pairs of qubits are physically adjacent or connected. A CNOT gate, for example, can only be applied between qubits that have a direct physical connection. This network of connections is called the backend's "coupling map".[1]

The **transpiler** is the Qiskit component responsible for solving this puzzle. It is a sophisticated compiler that takes your ideal, abstract circuit and rewrites it into a new, *equivalent* circuit that adheres to the target backend's native gate set and coupling map. This process, the "Optimize" step, may involve adding SWAP gates to move quantum information between non-adjacent qubits or decomposing complex gates. The goal is to produce the most efficient possible circuit for the specific hardware, as shorter circuits with fewer gates are less susceptible to noise.[1]

The easiest way to use the transpiler is via a preset pass manager, which bundles a pre-configured sequence of optimization passes.

Python

```python
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager

# First, select a backend (details in the next step)
backend = service.least_busy(simulator=False, operational=True)

# Generate a pass manager configured for the selected backend at optimization level 1
pm = generate_preset_pass_manager(optimization_level=1, backend=backend)

# Run the transpiler on our Bell state circuit (qc)
isa_circuit = pm.run(qc)

# We can draw the new, transpiled circuit to see the changes
isa_circuit.draw(output='mpl')
```

The resulting isa_circuit (ISA stands for Instruction Set Architecture) is now ready for execution on the chosen hardware.

6.3 Selecting a Backend and Running the Job

With the service object initialized, you can query for available backends. A pragmatic approach for a first run is to select the least busy operational device.

Python

```python
# The service.least_busy() method finds a suitable backend that is online and has the shortest queue.
backend = service.least_busy(simulator=False, operational=True, min_num_qubits=2)
print(f"Selected backend: {backend.name}")
```

Now, we use the Sampler primitive again, but this time, it is initialized with the real hardware backend object. Note that the runtime Sampler is imported from qiskit_ibm_runtime.

Python

```python
from qiskit_ibm_runtime import SamplerV2 as Sampler

# 1. Initialize the Sampler with the real hardware backend
sampler = Sampler(backend=backend)

# 2. Submit the job, running the transpiled 'isa_circuit'
# Note: Runtime jobs can be monitored on the IBM Quantum website.
job = sampler.run([isa_circuit])

# 3. Retrieve the result (this may take some time due to queueing)
print(f"Job ID: {job.job_id()}")
result = job.result()
```

6.4 Analyzing Real-World Results: The Noise is the Signal

The final step is to plot the results from the real device and compare them to the ideal simulation.

Python

```python
# Extract and plot the counts from the real device execution
real_counts = result.data.c.get_counts()
plot_histogram(real_counts, title=f"Bell State on {backend.name}")
```

When viewing this histogram, two key differences from the simulator's output will be immediately apparent:

1. The probabilities for 00 and 11 will not be perfectly balanced.
2. There will be small, but non-zero, bars for the "wrong" outcomes: 01 and 10.

This is not a mistake in the code. This is a direct observation of **quantum noise**. Real quantum computers are sensitive analog devices. Environmental interference, imperfect gate operations, and errors in the measurement process (collectively known as SPAM errors) cause the quantum state to deviate from the ideal, leading to erroneous outcomes.[6]

This comparison between the clean simulated result and the noisy real-world result provides one of the most important lessons for a beginner quantum programmer. It demonstrates that the central challenge in the current era of quantum computing is not just designing clever algorithms, but also developing techniques to mitigate the effects of noise. The discrepancy between theory and reality, visible in these two histograms, is the primary focus of a vast amount of ongoing research and development in the field.

Section 7: A Complete Workflow Example: The Deutsch-Jozsa Algorithm

To consolidate all the concepts and demonstrate the full "Map, Optimize, Execute, Analyze" workflow on a non-trivial problem, this section implements the Deutsch-Jozsa algorithm. This algorithm was one of the first to show a provable speedup over any possible classical algorithm for a specific task, making it a landmark in the history of quantum computing.[30]

7.1 The Problem and the Quantum Advantage

The Deutsch-Jozsa problem is centered around a "black box" function, or **oracle**, which takes an n-bit binary string as input and produces a single bit (0 or 1) as output. The function is guaranteed to be one of two types [31]:

- **Constant**: The function returns the same value (either always 0 or always 1) for all possible inputs.
- **Balanced**: The function returns 0 for exactly half of its inputs and 1 for the other half.

The challenge is to determine whether the function is constant or balanced by making the minimum number of queries to the oracle. Classically, in the worst-case scenario, one would need to query the function 2^(n-1) + 1 times to be certain of the answer. For example, if you query half of the inputs and they all return 0, it is still possible that the very next query returns a 1, revealing the function to be balanced.[32]

The Deutsch-Jozsa quantum algorithm, however, can solve this problem with 100% certainty by making **only a single query** to the quantum version of the oracle. This exponential speedup is achieved by leveraging the principles of quantum superposition and interference.[30]

7.2 Building the Algorithm in Qiskit (Map & Optimize)

The implementation involves creating oracles for both a constant and a balanced function and then wrapping them in the main algorithm circuit. For this example, we will use a 3-bit input (n=3).

1. Map: Creating the Oracles

First, we define the oracles. The circuit will need n input qubits and one ancillary qubit for the output, for a total of n+1 qubits.

Python

```python
import numpy as np
from qiskit import QuantumCircuit

n = 3

# Create a BALANCED oracle
# This oracle flips the output qubit if the input is '101'
# and does nothing for other inputs, making it balanced.
balanced_oracle = QuantumCircuit(n + 1)
b_str = "101" # The secret bitstring

# Wrap the controls in X-gates to match the bitstring
for qubit in range(len(b_str)):
    if b_str[qubit] == '1':
        balanced_oracle.x(qubit)

# Apply a multi-controlled CNOT (Toffoli) gate
balanced_oracle.mcx(list(range(n)), n)

# Unwrap the controls
for qubit in range(len(b_str)):
    if b_str[qubit] == '1':
        balanced_oracle.x(qubit)

# Convert the oracle circuit into a reusable gate
balanced_oracle_gate = balanced_oracle.to_gate()
balanced_oracle_gate.name = "Balanced Oracle"

# Create a CONSTANT oracle
# This oracle always outputs 1, regardless of input
constant_oracle = QuantumCircuit(n + 1)
constant_oracle.x(n) # Flip the output qubit to 1
constant_oracle_gate = constant_oracle.to_gate()
constant_oracle_gate.name = "Constant Oracle"
```

The balanced oracle here is constructed to be balanced for a specific input pattern, while the constant oracle simply flips the output qubit to 1 every time.[31]

2. Map: Building the Main Algorithm Circuit

The algorithm itself involves preparing the qubits, applying the oracle, and then performing a final transformation before measurement.

Python

```python
# Create the Deutsch-Jozsa circuit using the balanced oracle
dj_circuit = QuantumCircuit(n + 1, n)

# Step 1: Apply Hadamard gates to the input qubits
dj_circuit.h(range(n))

# Step 2: Prepare the ancillary qubit in the |-> state
```

```
dj_circuit.x(n)
dj_circuit.h(n)
dj_circuit.barrier()

# Step 3: Apply the oracle (one query)
dj_circuit.append(balanced_oracle_gate, range(n + 1))
dj_circuit.barrier()

# Step 4: Apply Hadamard gates to the input qubits again
dj_circuit.h(range(n))
dj_circuit.barrier()

# Step 5: Measure the input qubits
dj_circuit.measure(range(n), range(n))

# Visualize the final circuit
dj_circuit.draw(output='mpl')
```

The structure of the circuit—Hadamards, oracle, Hadamards—is designed to cause interference. If the function is constant, the final state of the input qubits will constructively interfere to become |00...0⟩. If the function is balanced, they will destructively interfere at the |00...0⟩ state, meaning any other outcome is possible, but never all zeros.[31]

3. Optimize: Transpiling the Circuit

As before, if this circuit were to be run on real hardware, it would first need to be transpiled. The process is identical to the one used for the Bell state. For this demonstration, we will proceed with the ideal circuit for simulation.

7.3 Executing and Analyzing the Results (Execute & Analyze)

We will now execute the circuit for the balanced oracle on a local simulator and analyze the outcome.

Python

```
from qiskit_aer.primitives import Sampler
from qiskit.visualization import plot_histogram

# Use the local Aer Sampler
sampler = Sampler()

# Execute the job
job = sampler.run(dj_circuit, shots=1024)
result = job.result()

# Get the counts and plot the histogram
counts = result.quasi_dists.binary_probabilities()
plot_histogram(counts, title="Deutsch-Jozsa with Balanced Oracle")
```

The resulting histogram will show probabilities for various non-zero bitstrings (like 101, 010, etc.), but the probability for the all-zero string, 000, will be effectively zero. This correctly identifies the oracle as **balanced**.

If the experiment were repeated by replacing balanced_oracle_gate with constant_oracle_gate in the main circuit construction, the resulting histogram would show a single bar at 000 with 100% probability. This would correctly identify the function as **constant**.

Running this algorithm on a real device would again introduce noise, slightly populating the "forbidden" measurement outcomes. For the balanced case, a small probability for 000 would appear. For the constant case, small probabilities for other states would appear. However, the correct outcome would still be overwhelmingly dominant, demonstrating the power of the algorithm even in the presence of moderate noise.[31]

Section 8: Your Journey Forward: Resources and Key Takeaways

This report has provided a comprehensive, step-by-step foundation for programming with Qiskit. The journey from a complete beginner to a proficient quantum developer is one of continuous learning and exploration. This final section summarizes the key skills acquired and provides a curated list of high-quality resources to guide your next steps into the expansive world of quantum computing.

8.1 Exploring Further: The Qiskit Universe

The Qiskit community and IBM provide a wealth of materials to support learners at all levels. The following resources are highly recommended for deepening your understanding and expanding your skills.

- **Official Qiskit Documentation and Tutorials**: The official documentation (quantum.ibm.com/docs) is the primary source of truth for all Qiskit features, APIs, and guides. The tutorials section contains numerous examples covering a wide range of topics from foundational concepts to advanced applications.[5]
- **Qiskit YouTube Channel**: The official YouTube channel (@qiskit) is an invaluable resource, featuring the "Coding with Qiskit" video series that walks through concepts and code, as well as recordings of seminars and summer schools on advanced topics.[2]
- **The Qiskit Textbook**: For those seeking a structured, university-course-level approach, the online Qiskit textbook offers in-depth explanations of quantum computation and algorithms, complete with interactive Jupyter notebooks that can be run directly on the IBM Quantum platform.[36]
- **Community Blogs and Forums**: The Qiskit blog on Medium and other community platforms feature articles and tutorials from developers and researchers, offering practical insights and diverse perspectives on using Qiskit to solve real problems.[9]
- **Advanced Algorithms and Applications**: Once comfortable with the basics, you can explore more complex topics. Excellent Qiskit tutorials exist for canonical algorithms like Quantum Teleportation [37] and for application domains such as Quantum Machine Learning.[39]

8.2 Key Takeaways: What You've Accomplished

By working through this report, you have progressed from having no prior Qiskit knowledge to successfully implementing a complete quantum algorithm. The core skills you have acquired form a solid foundation for any future work in the field:

- **Environment Mastery**: You can now set up a stable, professional, and isolated development environment for Qiskit using industry best practices (Anaconda and virtual environments).
- **Workflow Fluency**: You understand and can apply the four-step "Map, Optimize, Execute, Analyze" workflow, providing a structured mental model for tackling any quantum programming task.
- **Circuit Construction**: You can build quantum circuits from their fundamental components: qubits, classical bits, and essential quantum gates like the Hadamard, X, and CNOT.
- **Execution on Simulators and Hardware**: You are able to execute quantum programs on both ideal local simulators for testing and on real cloud-based quantum hardware for authentic, real-world results.
- **Result Interpretation**: You can visualize and interpret the output of a quantum program using histograms and, critically, you understand the fundamental difference between ideal, noiseless simulation results and the noisy outcomes from today's quantum computers.
- **Algorithmic Implementation**: You have successfully implemented the Deutsch-Jozsa algorithm, a true quantum algorithm that demonstrates a computational advantage, from start to finish.

The journey into quantum computing is both challenging and rewarding. Armed with these foundational skills and the knowledge of where to look for further learning, you are now well-equipped to continue exploring this exciting frontier of technology.

Works cited

1. Qiskit - Wikipedia, accessed August 4, 2025, https://en.wikipedia.org/wiki/Qiskit
2. Qiskit - YouTube, accessed August 4, 2025, https://www.youtube.com/qiskit
3. So What Is Qiskit Runtime, Anyway? - Medium, accessed August 4, 2025, https://medium.com/qiskit/so-what-is-qiskit-runtime-anyway-c78aecf3742
4. What is Qiskit? - YouTube, accessed August 4, 2025, https://www.youtube.com/watch?v=P5cGeDKOIP0
5. Introduction to Qiskit | IBM Quantum Documentation, accessed August 4, 2025, https://quantum.cloud.ibm.com/docs/guides
6. Qiskit Backends: A Technical Exploration | by Srila P. | Medium, accessed August 4, 2025, https://medium.com/@srilaa/qiskit-backends-a-technical-exploration-32f5f3f879ae
7. Hello world | IBM Quantum Documentation - IBM Quantum Platform, accessed August 4, 2025, https://quantum.cloud.ibm.com/docs/en/tutorials/hello-world
8. Qiskit - IBM Quantum Computing, accessed August 4, 2025, https://www.ibm.com/quantum/qiskit

9. A Beginners' Guide to Qiskit 1.0. This article provides an overview of ..., accessed August 4, 2025, https://medium.com/@harini.hapuarachchi/a-beginners-guide-to-qiskit-1-0-c8e3e854d732
10. Get backend information with Qiskit | IBM Quantum Documentation, accessed August 4, 2025, https://quantum.cloud.ibm.com/docs/guides/get-qpu-information
11. qiskit - PyPI, accessed August 4, 2025, https://pypi.org/project/qiskit/
12. Qiskit Backends: what they are and how to work with them | by Qiskit ..., accessed August 4, 2025, https://medium.com/qiskit/qiskit-backends-what-they-are-and-how-to-work-with-them-fb66b3bd0463
13. 3. Visualizing Quantum Measurements and States - Qiskit Pocket Guide [Book] - O'Reilly Media, accessed August 4, 2025, https://www.oreilly.com/library/view/qiskit-pocket-guide/9781098112462/ch03.html
14. Qiskit Installation Guide - CSE - IIT Kanpur, accessed August 4, 2025, https://www.cse.iitk.ac.in/users/rmittal/reports/Qiskit.pdf
15. Qiskit installation - Quantum Computing Stack Exchange, accessed August 4, 2025, https://quantumcomputing.stackexchange.com/questions/15431/qiskit-installation
16. Setting Up Qiskit On Your Computer | by Madeline Farina | QubitCo - Medium, accessed August 4, 2025, https://medium.com/qubitco/setting-up-qiskit-on-your-computer-b84cd63884c2
17. Build a simple Quantum Circuit using IBM Qiskit in Python - GeeksforGeeks, accessed August 4, 2025, https://www.geeksforgeeks.org/python/build-a-simple-quantum-circuit-using-ibm-qiskit-in-python/
18. #Steps to Install IBM Qiskit - QUANTUM VIDHYA, accessed August 4, 2025, https://quantumvidhya.com/steps-to-install-ibm-qiskit/
19. qiskit-tutorials/INSTALL.md at master - GitHub, accessed August 4, 2025, https://github.com/Qiskit/qiskit-tutorials/blob/master/INSTALL.md
20. Construct circuits | IBM Quantum Documentation, accessed August 4, 2025, https://quantum.cloud.ibm.com/docs/guides/construct-circuits
21. qiskit-community-tutorials/Coding_With_Qiskit/ep3_Hello_World ..., accessed August 4, 2025, https://github.com/Qiskit/qiskit-community-tutorials/blob/master/Coding_With_Qiskit/ep3_Hello_World.ipynb
22. Basic Circuit:Qiskit - Quantum Computing | ShareTechnote, accessed August 4, 2025, https://www.sharetechnote.com/html/QC/QuantumComputing_BasicCircuit_Qiskit.html
23. Getting Started with Qiskit (explained by a software developer, not a quantum physicist), accessed August 4, 2025, https://medium.com/queen-of-qiskit/getting-started-with-qiskit-explained-by-a-software-developer-not-a-quantum-physicist-b8ba820685e5

24. Visualize circuits | IBM Quantum Documentation, accessed August 4, 2025, https://quantum.cloud.ibm.com/docs/guides/visualize-circuits
25. Visualize results | IBM Quantum Documentation, accessed August 4, 2025, https://docs.quantum.ibm.com/guides/visualize-results
26. Qiskit: Histogram Tutorials - Deep Learning University, accessed August 4, 2025, https://deeplearninguniversity.com/qiskit/qiskit-histogram/
27. Hello World in Qiskit - Q-munity - The Quantum Insider, accessed August 4, 2025, https://qmunity.thequantuminsider.com/2024/06/10/hello-world-in-qiskit/
28. Hello World in Quantum Computing. Writing our first quantum program. - Medium, accessed August 4, 2025, https://medium.com/@netxspider/hello-world-in-quantum-computing-13d70c3055b1
29. Example: Running Jobs on Qiskit Backends - True-Q, accessed August 4, 2025, https://trueq.quantumbenchmark.com/guides/run/qiskit_submission_example.html
30. Mastering Deutsch-Jozsa Algorithm - Number Analytics, accessed August 4, 2025, https://www.numberanalytics.com/blog/deutsch-jozsa-algorithm-ultimate-guide
31. Learn-Quantum-Computing-with-Qiskit/Deutsch_Jozsa_Algorithm ..., accessed August 4, 2025, https://github.com/MonitSharma/Learn-Quantum-Computing-with-Qiskit/blob/main/Deutsch_Jozsa_Algorithm.ipynb
32. Implement the Deutsch-Jozsa Algorithm in Qiskit and IBM Quantum | by Hoa Nguyen (hoaio), accessed August 4, 2025, https://hoaio.medium.com/implement-the-deutsch-jozsa-algorithm-in-qiskit-and-ibm-quantum-98d7c12e87f6
33. Deutsch-Jozsa algorithm explained - Institute for Future Technologies, accessed August 4, 2025, https://ift.devinci.fr/quantum-computing-deutsch-jozsa-algorithm
34. Introduction to Qiskit | Coding with Qiskit 1.x | Programming on Quantum Computers, accessed August 4, 2025, https://www.youtube.com/watch?v=Tk9LOL9--Y4
35. Run Quantum Circuits with Qiskit Primitives - YouTube, accessed August 4, 2025, https://www.youtube.com/watch?v=NTplT4WnNbk
36. Top 6 Resources to Learn Quantum Computing for Free - DEV Community, accessed August 4, 2025, https://dev.to/balapriya/useful-resources-to-learn-quantum-computing-414k
37. Quantum Teleportation in Python - GeeksforGeeks, accessed August 4, 2025, https://www.geeksforgeeks.org/python/quantum-teleportation-in-python/
38. Quantum Teleportation Demystified with a Quantum Computer | Paradoxes Ep. 08, accessed August 4, 2025, https://www.youtube.com/watch?v=KsvNsY4cVvE
39. Machine Learning Tutorials - Qiskit Machine Learning 0.8.3 - GitHub Pages, accessed August 4, 2025, https://qiskit-community.github.io/qiskit-machine-learning/tutorials/index.html