Q1: Which is the correct, most common, and best way to create way to create a Pauli operator for the case of 105 qubits?

1. [("ZX", [101, 104], 1.0), ("YY", [0, 3], -1 + 1j)], num_qubits=105)
2. op = SparsePauliOp.from_sparse_list([("XXZ", [0, 1, 4], 1 + 2j), ("ZZ", [1, 2], -1 + 1j)], num_qubits=5)
3. op = Pauli("ZX") + Pauli("(-1 + 1j)YY")
4. op = Operator(np.random.rand(105, 105), input_dims=[2, 3], output_dims=[2, 3])

Q2: Which is the correct way to use a SparsePauliOp operator in a quantum circuit?

op = SparsePauliOp.from_sparse_list([("XXZ", [0, 1, 4], 1 + 2j), ("ZZ", [1, 2], -1 + 1j)],

num_qubits = 5)

1. qc = QuantumCircuit(105)

   qc.append(op, [0, 5])

   qc.draw("mpl")

2. qc = QuantumCircuit(5)

   qc.op([0, 5])

3. Neither of the above because a sparse Pauli operator is an observable. It is not a unitary transformation that can be directly applied as a gate in a quantum circuit. You would prepare and run a circuit and then measure the expectation value of this observable on the output of your quantum circuit.

Q3a: Visualize Quantum Circuits

Let's build a quantum circuit, qc = QuantumCircuit(3, 3), which has 3 qubits and 3 classical bits.

We add some gates:

qc.x(1)

qc.h(range(3))

qc.cx(0, 1)

All of the methods below are valid for drawing the circuit.

Which one draws the circuit using color by default rather than black-and-white?

1. print(qc)
2. qc.draw()
3. qc.draw(output="mpl")
4. qc.draw(output="latex")
5. all of the commands as written above will result in non-black-and-white graphs

Q3b: Visualize Quantum Circuits

Continuing with the circuit, qc, which we built in Q3a, say we add a barrier to the circuit before measuring the qubits,

```
qc.barrier()
qc.measure(range(3), range(3))
```

Can we choose to not show the barrier when drawing the circuit,

and can we reverse the order that the qubits are displayed?

1. yes, qc.draw(output="mpl", reverse_barriers=True, plot_barriers=False)
2. no, we can hide any barriers, but not reverse the order in which the qubits are displayed.
3. yes, qc.draw(output="mpl", reverse_bits=True, plot_barriers=False)

Q4: Visualize Quantum Measurements

Say we create a circuit, run it through a pass manager to convert the gates to ones that the hardware can execute, i.e. create an isa_circuit, where isa == Instruction Set Architecture), and run it using the Sampler:

```
job = sampler.run([isa_circuit])
result = job.result()
```

How would we create a histogram of the resulting measurements from the default 4096 shots?

1. plot_histogram(result.get_counts())
2. plot_histogram(result[0].get_counts())
3. plot_histogram(result[0].data.meas.get_counts())

Q5: Visualize Quantum States

Say we create a circuit, add some gates, and wish to view the quantum state of the quantum system that the circuit holds:

```
qc = QuantumCircuit(2)
qc.h(0)
qc.crx(pi / 2, 0, 1)
psi = Statevector(qc)
```

Which of the following is NOT something we can visualize?

1. the state vector written in Dirac braket notation?
2. the corresponding density matrix
3. the Bloch sphere of the state, psi
4. two Bloch spheres displaying the state of each qubit in psi
5. a qsphere of psi

Q6: Construct Basic Quantum Circuits

Let's say we have two circuits, qc1 and qc2, and we would like to combine them into one circuit, with qc1 applied to the qubits before qc2.

```
qc1 = QuantumCircuit(3)
qc1.x(0)
qc1.y(1)
qc1.z(2)

qc2 = QuantumCircuit(5)
qc2.h(range(5))
```

Which one of the following will create the circuit that we want?

1. qc = qc1.compose(qc2)

2. qc = QuantumCircuit(5)
   qc.append(qc1, [0,1,2])
   qc.append(qc2, range(5))

3. qc = qc2.compose(qc1)

Q7: Construct Dynamic Circuits

The Qiskit SDK includes functionality for performing classical feedforward and control flow. These features are sometimes referred to collectively as "dynamic circuits."

Which are the four control flow constructs for classical feedforward that Qiskit v.2.0 supports?

1. while, for, match, if

2. Qiskit.if_test
   Qiskit.switch
   Qiskit.for_loop
   Qiskit.while_loop

3. QuantumCircuit.if_test
   QuantumCircuit.switch
   QuantumCircuit.for_loop
   QuantumCircuit.while_loop

Q8: Construct Parameterized Circuits

Let's create a parameterized circuit:

```
from qiskit.transpiler import generate_preset_pass_manager
from qiskit.circuit import Parameter

angle_theta = Parameter("theta")  # undefined number
angle_phi = Parameter("phi")  # undefined number

# Create and optimize the circuit
qc = QuantumCircuit(1)
qc.rx(angle_theta, 0)
qc.rx(angle_phi, 0)
qc = generate_preset_pass_manager(
        optimization_level=3, basis_gates=["u", "cx"]
    ).run(qc)
```

Let's say we're given a parameterized circuit, qc, which has been defined similarly to the circuit above.

How can we determine the number of parameters that the circuit has?

1. compute the length of qc.angles
2. compute the length of qc.parameters or check qc.num_parameters
3. check qc.num_ancillas
4. check qc.num_input_vars
5. check qc.num_vars

Q9: Transpile and Optimize Circuits

Real quantum devices are subject to noise and gate errors, so optimizing the circuits to reduce their depth and gate count can significantly improve the results obtained from executing those circuits.

The *'generate_preset_pass_manager'* function has one required positional argument, optimization_level, that controls how much effort the transpiler spends on optimizing circuits. This argument can be an integer taking one of the values 0, 1, 2, or 3.

Which statement is true?

1. Higher numbers mean more optimization, and usually no additional compile time

2. optimization_level = 0 means light optimization

3. optimization_level = 3 means max optimization,
   will likely take more time than optimization_level = 1,
   and will always produce a better circuit result than any other choice for optimization_level, as long as we set approximation_degree to a value < 1

4. optimization_level = 0 inserts SWAP gates if necessary,
   optimization_level = 1 tries to select the best qubits in the graph for layout/routing and optimizes single qubit gates,
   optimization_level = 2 does commutative cancellation of gates,
   optimization_level = 3 resynthesizes two qubit blocks, moves measurements around to avoid SWAPs, and removes gates before measurements that would not affect the measurements

Q10: Differentiate Execution Modes to Optimize Job Queuing

Let's say you want to run a Variational Quantum Algorithm such as the Quantum Approximate Optimization Algorithm (QAOA) on real quantum hardware, and you're trying to decide if you should use Job, Batch, or Session mode.

What are the downsides of using each of the execution modes?

True or False:

Job – will result in longer wall-clock time, because each run in the iteration on a QPU has to enter the queue which increases the wall-clock time, and instead of getting better each iteration, with a longer wall-clock time, the result could get worse due to device drift and could take much longer to converge.

True or False:

Batch - same comments as for Job above

True or False:

Session - if you're using the Open Plan (10 mins free per month), Session is only available to you for the simulator and not for real hardware.

Q11: Run Quantum Circuits with real hardware in the IBM Runtime provider

Qiskit introduced the Primitives interface to help developers focus more on innovation and less on data conversion.

The interface is optimized for the two most common tasks for quantum computers:

1) sampling quantum states (Sampler)

2) calculating expectation values (Estimator)

In the context of layers of abstraction, a 'primitive' is the smallest processing instruction for a given abstraction level.

There are two types of Qiskit primitives: the base classes, and their implementations. Most users will interact with provider implementations, not the base primitives.

Which of the following is not true?

1. Implementations of the Primitives base classes include:
    a. Qiskit Runtime primitives (EstimatorV2 and SamplerV2)
    b. StatevectorEstimator and StatevectorSampler – which use the simulator built into Qiskit
    c. BackendEstimatorV2 and BackendSamplerV2 – which can be used to “wrap” any quantum computing resource into a primitive

2. EstimatorV2 and SamplerV2 can be imported from qiskit_ibm_runtime

3. StatevectorEstimator and StatevectorSampler can be imported from qiskit_ibm_runtime

4. One benefit of Qiskit Runtime primitives is that you can take advantage of the latest advancements in error mitigation and suppression by toggling options such as the primitive's resilience_level, rather than having to build your own implementation of these techniques.

Q12: Set options

You can set options for the Sampler primitive, such as, using a fixed seed to get fixed results and setting a default number of shots, for example:

```
options = {"simulator": {"seed_simulator": 42}, "default_shots": 200}

sampler = Sampler(mode=backend, options=options)
```

Some of the options allow setting sub-options, and these are:

dynamical_decoupling, environment, execution, simulator, twirling

Others just allow the setting of one int value:

default_shots and max_execution_time

Q12A. If you set sub-options for any of the options above which offer sub-options, how do you specify the sub-options?

1. use a dictionary containing the name of the sub-option as a key, and the value as the value, such as:

```
options = {“dynamical_decoupling”: {“enable”: True},
                “environment”: {“log_level”: “DEBUG”},
                “execution”: {“init_qubits”: True}}
```

2. use dot notation such as:

```
options = {dynamical_decoupling.enable = True,
                environment.log_level=”DEBUG”,  execution.init_qubits = True}
```

Q12B. For the options that don't allow sub-options, such as max_execution_time, how do you set that value?

1. include the name of the option as a key in a dictionary, and use the int value as the value for that key:

```
sampler = Sampler(mode=backend, options = {"max_execution_time": 60}
```

2. add the option to the call to the Sampler object like this:

```
sampler = Sampler(mode=backend, max_execution_time=60)
```

Q13: Sampler and dynamical decoupling and twirling

When using the Sampler we can set options for dynamical decoupling and for twirling.

What is dynamical decoupling (DD) and twirling? Why would we want to enable those?

Which two of these statements are false?

1. Twirling is a powerful tool for simplifying noise models in quantum computing, making it easier to analyze, simulate, and correct errors in quantum algorithms and hardware.

2. Dynamical decoupling uses a series of pulses to decouple the system from the environment to increase coherence time which could lead to more accurate quantum computations.

3. Although qubits in the ground state are less susceptible to decoherence and it can be beneficial to let them be while they are known to be in this state, there is no option to skip applying dynamical decoupling here.

4. Twirling involves averaging a quantum operation with a set of random unitary gates, usually from the Pauli group or Clifford group.

5. Twirling can only be performed on all qubits, and cannot be done on the subset of instruction qubits in the circuit.

Q14: Understand the theoretical background of the Sampler

Which one of the following is False?

1. The Sampler samples outputs of quantum circuits by sampling from their classical output registers.

2. The Estimator, given combinations of circuits and observables, estimates expectation values of the observables.

3. The reference implementation for the Sampler abstraction is the StatevectorEstimator.

Q15: The Estimator run method takes a PUB tuple as an input

Which of the following is not a valid statement?

1. A PUB is a data structure, known as a Primitive Unified Bloc.
   PUBs are the fundamental unit of work a QPU needs to execute workloads.

2. For the Estimator primitive, the format of the PUB should contain at most four values, which is the same max number of values that the PUB for the Sampler primitive should contain.

3. estimator_pub = (transpiled_circuit, observables, params)

4. The four values making up the PUB for an estimator are:
   a. A single QuantumCircuit, which may contain one or more Parameter objects
   b. A list of one or more observables, which specify the expectation values to estimate
   c. A collection of parameter values to bind the circuit against, if the circuit has Parameter objects
   d. (Optionally) a target precision for expectation values to estimate

Q16: The Estimator and the Resilience Level

The Estimator allows turning on error mitigation via the resilience_level, and to fine-tune the error mitigation via resilience sub-options.

Multiple approaches can be taken to set the options, such as:

```
qiskit_ibm_runtime import EstimatorV2 as Estimator
estimator = Estimator(backend, options={"resilience_level": 2})
# or...
options = EstimatorOptions()
options.resilience_level = 2
options.resilience.zne_mitigation = True
options.resilience.zne.noise_factors = [1, 3, 5]
estimator = Estimator(backend, options=options)
# or...
estimator = Estimator(backend)
options = estimator.options
options.twirling.enable_gates = True
options.resilience.measure_mitigation = True
# or...
estimator = Estimator(backend)
estimator.options.twirling.enable_gates = True
estimator.options.resilience.measure_mitigation = True
```

Which one of the following statements is false?

1. Not all options are available for both primitives.
2. Not all methods work together on all types of circuits.
3. All error mitigation and suppression can be turned off by setting the resilience_level to 0
4. the option max_execution_time limits the amount of time that the QPU is dedicated to processing your job, and is ignored when using local testing mode because that mode does not use quantum time.
5. resilience_level = 2 includes the features of resilience_level = 1 plus Zero Noise Extrapolation (ZNE) and gate twirling
6. Dynamic circuits are supported on all QPUs and dynamic circuits are compatible with dynamic decoupling.

Q17: Understand the theoretical background

We use the Estimator to estimate expectation values of observables. Here is an example:

```
from qiskit.primitives import StatevectorEstimator as Estimator
...
job1 = estimator.run([(psi1, H1, [theta1])])

job2 = estimator.run(
  [
    (psi1, [H1, H3], [theta1, theta3]),
    (psi2, H2, theta2)
  ],
  precision=0.01
)
```

Based on the examples above, which one of the following statements is inaccurate?

1. In job1, we are going to compute the expectation value of the Hamiltonian operator H1 on the quantum state |psi1(theta1)>

2. In job2, five estimates will be computed

3. In job2, the optional precision value specified is different from the default precision value

4. The observables being referred to in the statement "estimate expectation values of observables" are H1, H2, and H3.

5. The observables must be unitary

6. The parameters theta1, theta2, and theta3 parameterize psi1 or psi2, rather than H1, H3, or H2

Q18: Retrieve previous experiment results (session/runtime)

IBM Quantum automatically stores results from every job run on a QPU for you to retrieve at a later date.

You can use this feature to continue quantum programs across kernel restarts and review past results.

You can use “service”, see below, to help you retrieve jobs, and you can use a couple of classes from qiskit_ibm_runtime along with Python's built-in JSON library to save jobs locally.

```
from qiskit_ibm_runtime import QiskitRuntimeService
service = QiskitRuntimeService()
```

Which of the following statements are true?

1. You can use service.job() to retrieve the most recent jobs, or you can filter by backend name, creation date, and more to retrieve jobs.

2. You can import the classes RuntimeEncoder and RuntimeDecoder to use along with json.dump to save results to disk and json.load to retrieve saved results from disk.

3. If you already know the job_id, you can use this code to retrieve the results:

   ```
   retrieved_job = service.job(job_id)
   retrieved_job.result()
   ```

4. retrieved_job.result() returns an object of type PrimitiveResult

Q19: Monitor jobs

There are several methods on "job", say, from making a call to sampler.run() or estimator.run(). For example, job.result(), job.job_id(), job.status(), job.cancel()

Which one of these statements is false?

1. It is possible to retrieve information about execution spans in the results metadata for Sampler, which indicates that the QPU processing definitely occurred in the reported execution span, though the limits of the time window are not as tight as possible

2. Job results from job.result() are available as the job is running; you do not have to wait until the job completes.

3. This call can be used to retrieve a job previously submitted:

   job = service.job(<job_id>),
   where you fill-in the unique job_id that you obtained by calling job.job_id()

4. The possible values of job.status() are: *Literal*['INITIALIZING', 'QUEUED', 'RUNNING', 'CANCELLED', 'DONE', 'ERROR'], where job is a RuntimeJobV2.

5. You can cancel a job from the IBM Quantum Platform dashboard either on the Workloads page or the details page for a specific workload, or you can use Qiskit to cancel a job via job.cancel().

Q20: Structure types in OpenQASM 3 programs

OpenQASM (open quantum assembly language), a machine-independent programming interface compatible with IBM® QPUs, is an imperative programming language for describing quantum circuits.

OpenQASM is the common interchange format among independent quantum software tools. It acts as the bridge between tools that handle circuit construction, transpilation, etc., allowing the developer to distribute these tasks across different tools.

The Qiskit SDK provides ways to convert between OpenQASM and the QuantumCircuit class. OpenQASM 3 offers types for qubits.

Which one of these is not correct?

1. qubit[size] name;
   declares a quantum register with size qubits.
   These are referred to as "virtual qubits".

2. Physical or hardware qubits are referenced by the syntax $[NUM], where [NUM] is a non-negative integer, such as $0, $1.

3. Like virtual qubits, physical qubits are global variables, but unlike virtual qubits, they do not require explicit declaration.

4. Physical qubits can be used in gate statements.

5. OpenQASM 3 includes a new type to represent classical angles: angle. This type is intended to make manipulations of angles more efficient at runtime, when the hardware executing the program does not have built-in support for floating-point operations.

6. duration is a type to express timing. duration values can be converted to float using the division operator.

7. The built-in standard library of OpenQASM 3 includes several gate definitions for convenience. These include the gates such as x, h, cx, the phase gate, and many others.

Q21: Interpret OpenQASM semantics for versions 2 and 3

In OpenQASM a program is essentially a collection of statements, and each must end with a semicolon (;) to mark its conclusion.

A few more rules can be stated as follows, except one is incorrect, which one?

1. Identifiers for registers, variables and other entities can:
   Begin with a lowercase letter.
   Include only alphanumeric characters and underscores (_).
   Not coincide with a reserved keyword.

2. Whitespace is not ignored; it matters; it changes the logic of the program. Quantum operations are affected by the formatting.

3. A series of statements can be written on separate lines for better readability, or they can be on the same line.

4. This is an example of a valid simple OpenQASM Program which acts on 2 qubits:

```
// PhotonQ Experiment: A Simple OpenQASM Program
OPENQASM 2.0; // Specifies the OpenQASM version
include "qelib1.inc"; // Includes the standard library

// Declaration of quantum and classical registers
qreg qubits[2];  // Defines a quantum register with two qubits
creg ans[2];     // Defines a classical register with two bits

// Quantum gate application
h qubits[0];  // Applies a Hadamard gate to the first qubit
x qubits[1];  // Applies a Pauli-X gate to the second qubit

// Measure the qubit and save the result in the classical bit
measure qubits[0] -> ans[0];   // Measurement command
measure qubits[1] -> ans[1];   // Measurement command
```

Q22: Interoperate different versions of OpenQASM with Qiskit

OpenQASM2, OpenQASM3, and the Qiskit SDK

In order to convert between OpenQASM representations of quantum programs and the QuantumCircuit class, you must 'import qiskit.qasm3'.

What does this 'qiskit_qasm3_import' package provide?

1. Two high-level functions for importing from OpenQASM 3 into Qiskit:

```
qiskit.qasm3.load(file_name)
qiskit.qasm3.loads(program_string)
```

and two high-level functions for exporting Qiskit code to OpenQASM 3:

```
qiskit.qasm3.dumps(qc)
f = open("my_file.txt", "w")

dump(qc, f)
f.close()
```

2. Two high-level functions for importing from OpenQASM 3 into Qiskit:

```
qiskit.qasm3.load(file_name)
qiskit.qasm3.loads(program_string)
```

3. One high-level function for importing from OpenQASM 3 into Qiskit, and one high-level function for exporting Qiskit code to OpenQASM 3

4. One high-level function for importing from OpenQASM 3 into Qiskit

Q23: OpenQASM 2 vs. OpenQASM 3

What is the difference between OpenQASM 2 and OpenQASM 3? Select all that apply.

1. OpenQASM 3 is a large expansion over the previous OpenQASM 2 specification.

2. OpenQASM 3 is fully backwards compatible with OpenQASM 2, with no adjustments required ever.

3. OpenQASM 3 is focused on enabling more dynamic and complex quantum programs that integrate classical control flow, whereas OpenQASM 2 primarily describes static quantum circuits.

4. Both OpenQASM 2 and OpenQASM 3 include mechanisms for explicit timing and the ability to embed pulse-level definitions, which are crucial for advanced quantum control techniques like dynamical decoupling and calibration optimization.

5. OpenQASM 3 refines how gates are defined, and reduces reliance on built-in gates like CX.

6. In OpenQASM 2 and OpenQASM 3, qubits are initialized to the |0> state.

Q24: Interact with the Qiskit IBM Runtime REST API

The Qiskit Runtime REST API allows you to run on quantum processing units (QPUs) using Qiskit Runtime primitives, a simplified interface for circuit execution powered by advanced runtime compilation, error suppression, and error mitigation techniques, as well as getting information about instances and QPUs you have access to.

Which of these statements are true?

1. You can make a request with cURL or Python

2. Authentication in the form of an IBM Cloud Identity and Access Management (IAM) bearer token is required with every call as an http header.

3. You can submit only one circuit in a single job

4. You can create a session, and in your next job, use the session id to run this job as part of the session.

5. In this code snippet from a sample program being submitted via the REST API, the Sampler primitive is being used:

```
headersList = {
  "Accept": "application/json",
  "Authorization": "Bearer <YOUR_BEARER_TOKEN>",
  "Service-CRN": "<YOUR_INSTANCE_CRN>"
  "Content-Type": "application/json"
}

payload = json.dumps({
  "program_id": "estimator",
  "backend": "ibm_brisbane",
  "params": {
  "pubs": [["OPENQASM 3.0; include \"stdgates.inc\"; bit[1] c; x $0; c[0] =
            measure $0;", "Z"]],
  "options": {"dynamical_decoupling": {"enable": True}},
  "version": 2,
  "resilience_level": 1}
})
```

Answer Key:

Q1: #1

answer 1 is valid because the dimensions of the operator matrix are 2**105 by 2**105, and we said that we have 105 qubits.

This is a big matrix, since 2**105 equals 40564819207303340847894502572032

Using SparsePauliOp is beneficial for dealing with large quantum systems because it

can represent and manipulate operators efficiently, even with hundreds of qubits,

as long as the number of non-zero Pauli terms is relatively small.

answer 2 is not valid; the operator's dimensions are (32, 32) rather than 2**105 by 2**105

answer 3 is not valid; we experienced QiskitError exceptions above

answer 4 is not valid; we experienced QiskitError exceptions above

Q2: #3

The answer is 3 because SparsePauliOp's are not applied directly to a circuit as a gate would be, but instead are applied to the output state of a circuit to compute the expectation of the observable. This is Quantum Mechanics math.

Q3a: #3

The answer is 3, qc.draw(output="mpl"), where “mpl” stands for matplotlib.

Q3b: #3

The answer is 'yes', number 3; the barrier is not displayed and the qubits are displayed in reverse order.

Q4: #3

The answer is #3. The other attempts do not work.

Q5: #3

The answer is 3, since the Bloch sphere can only display the state of one qubit, and our state psi consists of 2 qubits.

"A qsphere is a visualization tool used in quantum computing to represent the state of a multi-qubit system. It's a generalization of the Bloch sphere, which is used for single qubits." (Google AI)

Q6: #2

The answer is 2. The first option results in an error since qc1 has fewer qubits (3) than qc2 which has 5. The code works for the third option, but the order of the gates for each qubit is not what we what since it applies qc2 before qc1.

Q7: #3

The answer is 3. The functionality is provided via these 4 methods on the QuantumCircuit class.

Q8: #2

The answer is 2. The QuantumCircuit class does not have an attribute called "angles". QuantumCircuit does have the num_ancillas attribute, but that refers to a qubit count and not a parameter count.

Q9: #4

The answer is 4. Higher optimization levels generate more optimized circuits at the expense of compile time. Optimization level 0 means no optimization and is used to characterize hardware. The default value for approximation_degree is 1, which means the transpiler makes no errors in the decomposition in order to save gates and ease execution. Lower values of approximation_degree will result in lower output accuracy.

Q10: T/T/T

The answer is True/True/True. Note, I tried running the QAOA Tutorial (found in the IBM Q Cloud Platform Tutorials collection) on 127 qubits using the execution mode Session. I submitted the job, and it cancelled after using about 10s of QPU time, returning an error about "canceled", "too much time in between jobs". Later, I read that Session is not available when running a job on real hardware if you're on the Open Plan (free 10 mins per month). You can use Session with the Simulator.

Q11: #3

The answer is 3. StatevectorEstimator and StatevectorSampler are imported from qiskit.quantum_info.

Q12A: #1

The answer is 1. Option 2 results in a SyntaxError.

Q12B: #1

The answer is 1. Option 2 results in a TypeError.

Q13: #3 and #5

The two False statements are 3 and 5.

Q14: #3

The answer is #3. The reference implementation for the Sampler abstraction is the StatevectorSampler, not the the StatevectorEstimator.

Q15: #2

The answer is 2. For the Sampler primitive, the format of the PUB should contain at most three values, not four.

A sampler pub is a list or tuple of one to three elements that define the unit of work for the sampler.

These are:

1. A single QuantumCircuit, possibly parameterized.

2. A collection of parameter value sets to bind the circuit against if it is parametric.

3. Optionally, the number of shots to sample.

Q16: #6

The answer is 6. Dynamic circuits are not supported on all QPUs and dynamic circuits are not compatible with dynamic decoupling.

See this table for feature compatibility:

https://quantum.cloud.ibm.com/docs/en/guides/runtime-options-overview.

Q17: #5

The answer is 5, the observables themselves do not have to be unitary. Note on answer #3, if precision is not specified, the estimator's default precision value will be used, which is sqrt(1/4096) = 0.015625.

Q18: #1, #2, #3, #4

The answer is all four of the statements are true.

Q19: #2

The answer is #2. The results are not available until the job completes.

Q20: #4

The answer is 4. Using physical qubits in gate statements is disallowed because it would make the code less portable and require specific hardware knowledge at the circuit definition level. Note on #3, virtual qubits are a higher-level abstraction that can be mapped to physical qubits by a compiler.

Q21: #2

The answer is 2 is not correct. Whitespace doesn't matter.

Q22: #1

The answer is 1.

Q23: #1, #3, #5

The answer is that the statements in 1, 3, and 5 are true.

Statement 2 is not true because there are some cases where adjustments are required, though generally OpenQASM 3 is backward compatible with OpenQASM 2.

Statement 4 is not true, as the statement only applies to OpenQASM 3.

Statement 6 is not true, since in OpenQASM 3 the qubits are initialized to an undefined state and must be explicitly 'reset' to ensure the |0> state.

Q24: #1, #2, #4

The answer is that statements 1, 2 and 4 are correct.

Statement 3 is not correct because you can submit more than one circuit in a single job.

Statement 5 is not correct because the Esimator primitive, rather than the Sampler primitive, is being used, as we see in the line: "program_id": "estimator",