

AutomationGlance Quantum Simulator — Circuits, DSL & Modeler Guide

Author of the original guide: Anup Dharangutti

Documentation version: 2.0

Compatible with: current AutomationGlance simulator

Last reviewed: 2026-10-05

Generated from the canonical web documentation. Software behavior is version-dependent; review this guide when the implementation changes.

[Documentation index](#) · [Download PDF](#)

1. [docs/dsl](#)
2. [docs/circuits](#)
3. [quantum-modeler/methodology](#)
4. [docs/worked-examples](#)
5. [docs/architecture](#)
6. [docs/openqasm](#)
7. [api](#)
8. [docs/where-it-fits](#)
9. [docs/references](#)

Quantum Circuit DSL Reference

The authoritative reference for the circuit text accepted by the AutomationGlance simulation API.

Documentation version: 2.0 · Compatible with: current AutomationGlance simulator · Last reviewed: 2026-10-05

Declaration and line structure

Declare `qubits N` with an integer `N` from 1 to 8 for execution. Use one statement per line. Blank lines and lines whose first non-whitespace character is `#` are ignored. Inline comments and semicolon-separated statements are not supported. Commands and gate names are case-insensitive. Leading/trailing whitespace and repeated spaces or tabs between tokens are accepted.

The declaration is required, but the parser does not require it to be first. Put it first to make bounds checks clear. Repeated declarations produce `qubits_redeclared` warnings; the last value sets execution width. Gates are checked against the declaration in effect on their line and again against the final width at execution. CR and LF are each split by the parser, so reported line numbers for CRLF input can differ from editor line numbers. Prefer LF.

Supported gates

Public DSL gate signatures (qubits before angles)

Syntax	Meaning
<code>H i, X i, Y i, Z i</code>	Hadamard and Pauli gates
<code>S i, T i</code>	Phase gates $\text{diag}(1, i)$ and $\text{diag}(1, \exp(i\pi/4))$
<code>RX i angle, RY i angle, RZ i angle</code>	$\exp(-i \text{angle } P/2)$, for $P = X, Y, Z$
<code>U3 i theta phi lambda</code>	Matrix rows: $[\cos(\theta/2), -\exp(i\lambda)\sin(\theta/2)]$ and $[\exp(i\phi)\sin(\theta/2), \exp(i(\phi+\lambda))\cos(\theta/2)]$
<code>CX c t</code> or <code>CNOT c t</code>	Controlled X; control first, target second
<code>CZ c t</code>	Controlled Z

Gate indices are zero-based integers in $[0, N-1]$. Gate operands also accept a `q` prefix, such as `H q0`. Parenthesized, comma-separated gate calls such as `RX(0, 1.5707963267948966)` and `CNOT(q0, q1)` are accepted. This is qubit-first DSL syntax, unlike OpenQASM. Empty comma

entries are ignored by the backend parser; avoid them because the browser exporter rejects them. Control and target must be distinct; equal indices are rejected with `gate_qubits_distinct`.

Numbers and angles

Angles are radians, parsed with invariant decimal notation: integers, signed decimals, leading decimal points, and scientific notation (for example `-1.25e-2`). Use numeric values for π ; expressions such as `pi/2`, fractions, degree suffixes and decimal commas are unsupported. Qubit counts and indices use signed 32-bit integer parsing; positive signs and leading zeros are accepted, but fractional and negative indices are invalid. Nonfinite values such as NaN, Infinity and overflowing exponents are rejected with `gate_parameter_invalid`.

Measurement is terminal sampling

`MEASURE i` takes one integer index; it does not accept `q0` or parentheses. Measurements are collected in source order and performed after *all* unitary gates, even if written earlier. There is no mid-circuit collapse, classical control, reset or conditional feedback. Repeated measurement indices repeat the same sampled bit. With no `MEASURE` statements, the response has an empty counts object but still includes the full statevector and probabilities.

State labels use `q0` at the left (the most significant statevector index bit). Counts contain only the measured bits in declaration order: preparing `q0=1,q1=0` and measuring 1 then 0 returns `01`; the full state is `10`.

Working examples

Bell pair

```
qubits 2
H 0
CX 0 1
MEASURE 0
MEASURE 1
```

Full-state probabilities are $\frac{1}{2}$ for `00` and `11`. Sampled counts fluctuate with shots and seed.

Parameterized and alternate gate notation

```
# Angles are radians
qubits 2
ry(q0,1.5707963267948966)
RZ 0 -1.25e-2
U3 1 0.5 0 1e-1
CZ 0 1
MEASURE 1
MEASURE 0
```

Validation and unsupported syntax

Unknown gates, wrong arity, missing declarations and invalid indices yield parse errors and HTTP 422; errors include a code and, where available, a line. Width above 8 is rejected at execution with HTTP 422. See [API request limits and error formats](#). The editor's local analysis is advisory; the API validates execution. No auto-expansion of undeclared registers occurs.

```
H 0
```

```
qubits 2  
SWAP 0 1
```

```
qubits 2  
MEASURE q0
```

```
qubits 2  
RX 0 pi/2
```

```
qubits 2  
H 2
```

The older `qubit q0` declaration, named registers, SWAP, CY, CCX, custom gate definitions, OpenQASM source, loops and conditions are unsupported. SWAP can be decomposed into three CX operations on distinct qubits; it is not a public DSL instruction.

Reproducibility

The DSL has no shots or seed statements. Those are API fields. A supplied signed 32-bit seed controls sampling; otherwise the simulator API hashes the exact circuit text and shots. Whitespace changes can therefore change the default counts without changing probabilities. Reproducibility assumes the same backend/runtime version. See [OpenQASM export limitations](#).

Circuit Examples and Explanations

Version-aligned Bell, GHZ, teleportation, Deutsch–Jozsa, Grover, phase and parity-check demonstrations.

Documentation version: 2.0 · Compatible with: current AutomationGlance simulator · Last reviewed: 2026-10-05

All examples use the current DSL, zero-based indices and terminal measurement. Each is executed in documentation regression tests.

Bell state

```
qubits 2
H 0
CX 0 1
MEASURE 0
MEASURE 1
```

Measurements of the two qubits exhibit correlated outcomes in the computational basis: 00 and 11 each have probability $\frac{1}{2}$. The known ideal circuit prepares $(|00\rangle + |11\rangle)/\sqrt{2}$; correlation alone is not an entanglement witness.

GHZ state

```
qubits 3
H 0
CX 0 1
CX 0 2
MEASURE 0
MEASURE 1
MEASURE 2
```

Computational-basis measurements yield correlated outcomes 000 and 111, each with ideal probability $\frac{1}{2}$. No faster-than-light communication follows.

Teleportation preparation and Bell measurement

```
qubits 3
H 1
CX 1 2
H 0
CX 0 1
H 0
MEASURE 0
MEASURE 1
```

q0 contains $|+\rangle$; q1 and q2 form the resource pair. The final CX and H implement the Bell-basis change. Alice's two output bits are uniformly distributed. This is a partial protocol: the DSL has no measurement-conditioned X/Z corrections and cannot complete arbitrary-state teleportation through classical feedback. Adding unconditional X/Z gates is not a replacement.

Deutsch–Jozsa: one-input-bit balanced oracle

```
qubits 2
X 1
H 0
H 1
CX 0 1
H 0
MEASURE 0
```

The oracle $f(x)=x$ is balanced. Phase kickback and the final H make the measured input bit 1 with probability 1. A constant oracle would produce 0. This demonstrates a query-model distinction; a tiny classical simulation is not a practical quantum speedup.

Two-qubit Grover search

```
qubits 2
H 0
H 1
CZ 0 1
H 0
H 1
X 0
X 1
H 1
CX 0 1
H 1
X 0
X 1
H 0
H 1
MEASURE 0
MEASURE 1
```

CZ marks $|11\rangle$ with a minus sign; the remaining gates implement diffusion up to a global phase. One iteration yields 11 with ideal probability 1.

Superposition sampler

```
qubits 2
H 0
H 1
MEASURE 0
MEASURE 1
```

The product state $|++\rangle$ has probability $\frac{1}{4}$ for each of 00, 01, 10 and 11. Counts approach those probabilities over many shots.

A fixed random-style circuit

```
qubits 2
H 0
CX 0 1
RZ 1 1.234
X 0
H 1
MEASURE 0
MEASURE 1
```

A fixed example for exploring phase and interference. This particular circuit has uniform computational-basis probabilities despite nontrivial phases. Random circuit generation and shot-sampling seed are separate controls; random circuits do not establish a hardware benchmark.

Phase kickback

```
qubits 2
H 0
X 1
CZ 0 1
H 0
MEASURE 0
```

With target q1 in $|1\rangle$, CZ applies a Z phase to the control. The final H converts that phase into a deterministic $q_0=1$ result.

QFT context: phase intuition

```
qubits 1
H 0
S 0
T 0
H 0
MEASURE 0
```

This is a phase-interference demonstration, not a full quantum Fourier transform: $P(0)=(2-\sqrt{2})/4$ and $P(1)=(2+\sqrt{2})/4$. QFT uses controlled phase rotations and final wire ordering. Those rotations can be decomposed into supported gates, but there is no native QFT or controlled-phase DSL instruction. The v1 guide's CZ/RZ/SWAP listing was not a valid implementation of QFT; it has been replaced by this explicitly limited demonstration.

Repetition-code parity checks

```
qubits 5
H 0
CX 0 1
CX 0 2
# Inject X on the first encoded data qubit
X 0
# Ancillas record adjacent parity checks
CX 0 3
CX 1 3
CX 1 4
CX 2 4
MEASURE 3
MEASURE 4
```

The first three qubits encode a repetition state. After the deliberate X0 error, ancillas q3,q4 give syndrome 10 with probability 1. With no injected error the syndrome is 00; X1 yields 11 and X2 yields 01. This illustrates detection under a single-bit-flip assumption, not protection from arbitrary quantum errors or implemented conditional correction.

Reading results

Gate count and greedy depth describe structure, not physical execution time. The interaction graph indicates controlled-gate pairs; it is not a state-based entanglement test. Amplitudes carry complex phase; squared magnitudes give probabilities. A reduced Bloch vector can lie inside the sphere for an entangled pure joint state. See [analysis limitations and Complexity Score](#).

Using the simulator

Run sends a request to the API. Reset and sample loaders prepare editor contents. Random circuit generation provides exploratory examples. Share stores encoded circuit/settings in a link. Copy QASM exports a supported OpenQASM 2.0 subset. SVG/PNG downloads require a current rendered diagram. Performance mode reduces UI effects, not simulation memory requirements. API Playground exercises the same endpoint.

Quantum Modeler Methodology

The implemented formulations, bounded QAOA search, exact classical checks, and resource projections.

Documentation version: 2.0 · Compatible with: current AutomationGlance simulator · Last reviewed: 2026-10-05

Execution and formulation

Max-Cut, Knapsack and Custom QUBO use ideal, noiseless classical statevector simulation with at most 8 qubits. Each executes $p=1$ QAOA and performs a separate exhaustive classical check. No quantum hardware or quantum advantage is demonstrated. [Worked examples](#) make the algebra independently checkable.

Binary variables and Ising convention

$x_i \in \{0,1\}$; Z_i has eigenvalue $+1$ on $|0\rangle$ and -1 on $|1\rangle$. Thus $x_i = (1-Z_i)/2$. This maps classical assignments to computational-basis states.

QUBO representation

$E(x) = c + \sum_i a_i x_i + \sum_{i<j} b_{ij} x_i x_j$ is minimized. The constant changes reported energies but not minimizing assignments. Since $x_i^2 = x_i$, diagonal matrix coefficients belong in the linear array. Pair coefficients occur once; the input is a sparse coefficient list, not a full matrix.

Custom QUBO accepts one coefficient per variable, up to 28 distinct pairs, and finite coefficients/constant of magnitude at most 1,000,000. A reversed pair is normalized to $i<j$; supplying both (i,j) and (j,i) , or any duplicate, is rejected, not summed. Diagonal pair entries are rejected. To convert a matrix $x^T Q x$, use $a_i = Q_{ii}$ and $b_{ij} = Q_{ij} + Q_{ji}$. A symmetric matrix therefore contributes twice each off-diagonal entry.

AutomationGlance validates the QUBO that you provide. It cannot infer unstated business constraints. Constraints must be encoded in the QUBO, typically using appropriate penalty terms. Custom QUBO is unconstrained binary minimization; it does not attach a separate business-feasibility meaning to an assignment.

Auditable QUBO → Ising mapping

$x_i x_j = (1 - Z_i - Z_j + Z_i Z_j)/4$. Substitution gives $H = C \cdot I + \sum_i h_i Z_i + \sum_{i<j} J_{ij} Z_i Z_j$, with

- $C = c + \frac{1}{2} \sum_i a_i + \frac{1}{4} \sum_{i<j} b_{ij}$

- $h_i = -a_i/2 - 1/4 \sum_{j \neq i} b_{(ij)}$ (each incident pair once)
- $J_{ij} = b_{ij}/4$.

Each basis eigenvalue of H equals the original $E(x)$. Compilation divides nonconstant coefficients by $s = \max(1, \max|h_i|, \max|J_{ij}|)$. This preserves ground states. Reported energies use the original, unscaled QUBO. The constant's global phase is omitted from the circuit.

Max-Cut

The graph is an unweighted simple undirected graph with 2–8 executed vertices. Edges have distinct in-range endpoints; duplicate undirected edges are rejected. Empty graphs are permitted. Bit x_i selects the partition of vertex i ; an edge is cut if its endpoint bits differ.

Maximize $C(x) = \sum_{(ij) \in E} (x_i + x_j - 2x_i x_j)$. The cost Hamiltonian is $H_c = \sum_{(ij) \in E} (I - Z_i Z_j)/2$. The product uses this *maximization* convention directly, rather than silently minimizing C . One cost interaction $\exp(-iy(I - Z_i Z_j)/2)$, ignoring global phase, compiles to $CX(i, j)$, $RZ(j, -y)$, $CX(i, j)$. Measured bitstrings are candidate graph partitions.

Knapsack

Select items x_i to maximize $\sum v_i x_i$ with $\sum w_i x_i \leq C$. Weights are integers 1–127, values 1–1000, capacity 1–127, with 1–7 items and total encoded width ≤ 8 . Duplicate items are distinct optional selections. Binary slack s_k has weights 2^k for $k = 0 \dots \text{floor}(\log_2 C)$, so $q = n + \text{floor}(\log_2 C) + 1$.

The minimized encoding is $E(x, s) = -\sum v_i x_i + A(\sum w_i x_i + \sum 2^k s_k - C)^2$, with fixed $A = \sum v_i + 1$. This is expanded using $x^2 = x$. With combined weights $d = (w_0, \dots, 1, 2, 4, \dots)$, the constant is AC^2 , linear coefficients are $A(d_i^2 - 2Cd_i)$ minus the item value (zero for slack), and pair coefficients are $2Ad_i d_j$.

For integer inputs, any nonzero equality residual costs at least A , exceeding all possible item rewards. Every feasible item selection can represent its remaining capacity as slack. Thus the exact encoded ground state maximizes feasible value. A feasible selection ($\text{weight} \leq C$) can still have the wrong slack and a positive penalty; `feasibleProbability` and `constraintSatisfiedProbability` measure different events. The reported conditional expected feasible value averages only over feasible mass.

Penalty formulations depend on suitable coefficients. Here the fixed A has an integer-input guarantee for exact ground states, but a shallow circuit can sample infeasible assignments and large penalties can obscure value differences. No-feasible-sample results keep the measured best and approximation ratio null; the classical answer is never substituted. A zero exact value also has a null ratio.

QAOA: one layer and bounded grid search

Start from $|0\dots 0\rangle$ and apply H to every qubit to prepare $|+\rangle^n$. The ansatz is $\exp(-i\beta\sum X_i)\exp(-iyH_c)|+\rangle^n$. Apply cost first, then $RX(2\beta)$ on each qubit, then terminal computational-basis sampling. Parameters are numeric radians; only $p=1$ (one γ and one β) is supported.

For QUBO/Knapsack, the normalized cost uses $RZ(2\gamma h_i/s)$ and $CX-RZ(2\gamma J_{ij}/s)-CX$. Max-Cut uses the sign described above without this QUBO normalization.

Implemented angle search

Model	Grid	Selection
Max-Cut	$\gamma=k\pi/8, \beta=l\pi/8;$ $k,l=0\dots 7$	Maximize exact expected cut; improvements $>10^{-12}$
Knapsack / QUBO	$\gamma=k\pi/4, \beta=l\pi/8;$ $k,l=0\dots 7$	Minimize original expected energy; improvements $>10^{-10}$

At most 64 combinations are evaluated, beta varying first; ties retain the first pair.

`QuantumModel.__MaxEvaluations` clamps the budget to 1–64 and selects a prefix of that order. Search uses full state probabilities, not noisy shot estimates. This is a fixed grid, not an adaptive variational optimizer. Only the final selected circuit is sampled (1–100,000 shots); modeler seed defaults to 42. An explicit seed and the same backend/runtime reproduce counts; seed does not change the selected angles.

Classical validation

Independent exhaustive enumeration checks at most 256 assignments. Max-Cut enumerates all partitions; Knapsack enumerates item selections with direct capacity checks; Custom QUBO enumerates its supplied energy. These establish the optimum for the supplied bounded model, not an unstated business problem or a projected large instance.

Custom QUBO reports the raw minimum and ground states within tolerance $10^{-9}+10^{-14}(|c|+\sum|a|+\sum|b|)$. Its additive optimality gap is best sampled energy minus exact minimum; ratios are inappropriate for arbitrary signed energies. Max-Cut reports best sampled cut/exact cut, with a null ratio when the exact cut is zero. A sampled optimum is not guaranteed.

Executed, projected, logical and physical scale

EXECUTED means an allocated statevector of width ≤ 8 . **PROJECTED** means symbolic resource arithmetic on requested target sizes; no target graph, solution or statevector was constructed. A projection being small enough to execute does not mean it was executed.

- Max-Cut logical variables: N vertices; $p=1$ gate estimate $2N+3M$ when M target edges is supplied; otherwise interaction/gate counts are unknown.
- Knapsack: $q=n+\text{ceil}(\log_2(C+1))$; dense pair bound $q(q-1)/2$; gate bound $3q+3q(q-1)/2$.
- Custom QUBO: q target variables, m target pairs; gate upper bound $3q+3m$.

These counts include H preparation and cost/mixer gates, exclude measurement and hardware compilation, and do not establish physical circuit depth. QAOA layer count p is not hardware depth. Target topology, native gates, routing and parallel scheduling would be required for a physical depth estimate. No physical-qubit or error-correction overhead estimate is implemented.

An ideal dense complex-double statevector alone needs 16×2^q bytes: 8 qubits need 4 KiB; 30 need 16 GiB. Simulation also allocates work arrays and results. Symbolic 2^q strings for larger targets describe this exponential growth; they do not claim available memory or executable scale.

Bit ordering and artifacts

Logical assignments display $x_0 \dots x_{N-1}$ left to right; q_0 is the most significant bit of statevector indexing. Knapsack puts item bits first, then slack bits of weights $1, 2, 4, \dots$. For three modeler variables, 100 selects x_0 only. Exports measure $x_i \rightarrow q[i] \rightarrow c[i]$; Qiskit commonly displays a single classical register as $c[N-1] \dots c[0]$, yielding 001 for that same assignment. Reverse only after checking the actual register/measurement mapping; arbitrary subsets need explicit conversion. No external hardware or SDK execution is claimed.

All modelers export DSL and OpenQASM 2.0. Custom QUBO additionally exports `automationglance.qubo.v1` JSON, with objective/order/provenance metadata; extract coefficient fields when making a strict API request. Edits invalidate old exports until rebuilt. See [export compatibility](#) and [sources and implementation references](#).

Worked Max-Cut and QUBO Examples

Trace small models from objective and Hamiltonian to candidate assignments and exact classical verification.

Documentation version: 2.0 · Compatible with: current AutomationGlance simulator · Last reviewed: 2026-10-05

A. Triangle Max-Cut

Take vertices $\{0,1,2\}$ and edges $\{(0,1),(0,2),(1,2)\}$. Bit x_i indicates which side contains vertex i .

Maximize $C(x)=2x_0+2x_1+2x_2-2x_0x_1-2x_0x_2-2x_1x_2$.

Substitute $x=(I-Z)/2$: $H_c=3I/2-(Z_0Z_1+Z_0Z_2+Z_1Z_2)/2$. Equal-bit assignments 000 and 111 cut zero edges. Every other assignment cuts two; a triangle cannot have all three edges cross a binary partition. Enumeration of eight assignments proves optimum 2.

The actual $p=1$ circuit applies H on all three wires, then CX–RZ($-\gamma$)–CX for each edge, then RX(2β) on all wires, then measurements 0,1,2. With the full 64-point default search it selects $\gamma=\beta=\pi/8$. Use the triangle preset or this request to POST `/api/v1/quantum-model/maxcut` (add your sessionId when required):

```
{
  "vertices": 3,
  "edges": [
    [
      0,
      1
    ],
    [
      1,
      2
    ],
    [
      0,
      2
    ]
  ],
  "targetVertices": 3,
  "targetEdges": 3,
  "qaoaDepth": 1,
  "shots": 1024,
  "seed": 42
}
```

Observed local reference run: 1024 shots, seed 42,
.NET 10

Assignment	Cut	Ideal probability	Count
000	0	0.01987622	25

Assignment	Cut	Ideal probability	Count
001	2	0.16004126	184
010	2	0.16004126	146
011	2	0.16004126	161
100	2	0.16004126	164
101	2	0.16004126	155
110	2	0.16004126	171
111	0	0.01987622	18

Expected cut ≈ 1.92049513 ; optimal-assignment probability ≈ 0.96024756 . The best sampled cut is 2 and the displayed approximation ratio is $2/2=1$. The expected-cut/exact-optimum comparison is ≈ 0.96025 , a different statistic. Counts are a reproducible reference for this runtime, not a guaranteed distribution for another seed or backend. Large target settings do not change this three-qubit execution.

B. Custom QUBO

Minimize $E(x) = -3x_0 - 2x_1 + 4x_0x_1$. This is an unconstrained supplied objective: the positive pair coefficient discourages selecting both, but no unspoken business rule is enforced.

Substitution gives $-3(I-Z_0)/2 - 2(I-Z_1)/2 + (I-Z_0-Z_1+Z_0Z_1)$, so $H = -1.5I + 0.5Z_0 + 0Z_1 + 1Z_0Z_1$. The Ising constant is -1.5 ; linear Z coefficients are $[0.5, 0]$; the ZZ coupling is 1. Energy scale is $\max(1, 0.5, 1) = 1$.

All assignments, with x0 leftmost

x0 x1	z0 z1	E(x)=H(z)
00	+1 +1	0
01	+1 -1	-2
10	-1 +1	-3 (unique optimum)
11	-1 -1	-1

Set two variables, constant 0, linear array $[-3, -2]$ and a single pair (0,1) with coefficient 4 in Custom QUBO. Equivalently send this to POST `/api/v1/quantum-model/qubo`:

```
{
  "variableCount": 2,
  "constant": 0,
  "linear": [
    -3,
    -2
  ],
  "quadratic": [
    {
      "i": 0,
      "j": 1,
      "coefficient": 4
    }
  ],
  "qaoaDepth": 1,
  "shots": 1024,
  "seed": 42
}
```

The circuit prepares $|++\rangle$, applies $RZ(\gamma)$ to q_0 and $CX(0,1)-RZ(2\gamma)$ on $q_1-CX(0,1)$, then $RX(2\beta)$ on each wire and terminal measurement. Constant phase is omitted. Exact validation reports minimum -3 and ground state 10 . The sampled candidate list is distinct from that reference; if 10 is observed its energy gap is zero, while a best sample of 01 has gap $(-2)-(-3)=1$.

Exported QUBO JSON retains the original coefficients; Ising fields report the derived expression. DSL and OpenQASM encode the selected-angle experiment, not a guarantee that every measurement is optimal. See [methodology and penalties](#).

Execution Architecture and Analysis

Where editing, simulation, diagrams, storage, and heuristic circuit analysis actually run.

Documentation version: 2.0 · Compatible with: current AutomationGlance simulator · Last reviewed: 2026-10-05

Execution architecture

1. **Browser:** DSL editing, syntax highlighting, input checks, request construction, local structure analysis and visualization.
2. **Azure Function API:** request/rate/usage validation, DSL parsing, ideal statevector calculation, Bloch vectors and probabilities, seeded shot sampling, and SVG circuit diagram generation.
3. **Browser:** displays returned results, renders the statevector explorer, copies DSL/OpenQASM, decodes returned SVG for standalone download and converts it to PNG.

Modeler API routes additionally build the formulation and circuit, search parameters, enumerate the bounded classical reference, and calculate symbolic resource projections. These computations are not performed solely in the browser.

Production and localhost UI configuration currently call the Azure Function host directly, as specified in `frontend/config/settings.js`. The website's `/api/` URL is a documentation page, not the simulation host. Running a simulation or building a model requires network access and an available API. Editing and inspecting already-loaded results do not require a new execution request. The startup text preview is a browser placeholder, not a simulated diagram.

Storage and sharing

The simulator saves editor/settings/theme state and a usage session identifier in browser `localStorage`; Learn progress uses `ag-learn-progress-v1`. Browser storage may persist between sessions and can be cleared by the user. Seed/settings are carried in modeler handoff URLs. Share URLs contain the encoded circuit and settings; encoding is not encryption. API calls transmit the circuit/model, execution settings and session identifier. Do not interpret browser-first as private, offline computation.

Entangling-gate interaction analysis

The browser tracks controlled gates and distinct qubit pairs. This indicator is derived from circuit operations and does not independently calculate an entanglement measure from the final quantum state. Entangling-capable gates do not necessarily leave a state entangled. Likewise

computational-basis correlations alone do not verify entanglement: a classical mixture can have the same probabilities. Protocol descriptions are circuit-pattern hints, not proofs.

AutomationGlance Complexity Score

The score is a normalized heuristic for comparing circuit structure inside AutomationGlance. It is not a standardized quantum-computing complexity measure, runtime prediction, hardware error rate, computational hardness estimate, or evidence of quantum advantage.

$S = \min(1, 0.4 \min(d/50, 1) + 0.3 \min(g/100, 1) + 0.2 \min(e/20, 1) + 0.1 \min(n/8, 1))$. Here d is greedy circuit depth (each operation takes one layer, including measurement), g is non-measurement gate count, e is the number of distinct controlled-gate interaction pairs (not the number of entangled states), and n is width. Gates on disjoint wires can share a layer. Measurements in local structure analysis retain written positions, although the API samples them terminally.

For valid supported input the expected range is 0–1, displayed to two decimals. Larger values mean more of these structural factors, saturating at the chosen thresholds; those thresholds are UI heuristics, not physical limits. Performance mode reduces UI animation and shadows; it does not increase the 8-qubit execution ceiling. See the [score implementation](#).

OpenQASM Export Compatibility

AutomationGlance exports a gate subset of OpenQASM 2.0; it does not import OpenQASM.

Documentation version: 2.0 · Compatible with: current AutomationGlance simulator · Last reviewed: 2026-10-05

What is supported

The simulator's Copy QASM action translates current DSL in the browser. Modeler artifacts are exported by the API. Both emit `OPENQASM 2.0; include "qelib1.inc";`, a `q` register, and a `c` register when measurements exist. The [public DSL gate set](#) maps to `h`, `x`, `y`, `z`, `s`, `t`, `rx`, `ry`, `rz`, `u3`, `cx` and `cz`; CNOT becomes `cx`. Numeric angles are radians. Unsupported gates cause an export error.

This is export only, not an OpenQASM interpreter or importer. OpenQASM 3, custom gates, classical conditions, reset, loops, opaque operations, hardware mapping and noise instructions are not supported. Exporting a circuit does not execute it in another SDK or on a QPU.

Measurement and ordering

All unitary gates precede exported measurements, matching DSL terminal sampling. Measurement statement number `k` maps its selected `q[i]` to `c[k]`. Comments record the mapping.

AutomationGlance displays measured bits in statement order, while tools may display the classical register in reverse order. Use [the bit-order example](#) before comparing results.

Example export

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c[2];
h q[0];
cx q[0],q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
```

Equivalent to the Bell example in the [DSL reference](#); exporter output also includes order comments. For best cross-export consistency, use one initial declaration, finite numeric angles, distinct two-qubit operands and no empty comma arguments. Export syntax validation is not a claim of tested SDK/hardware interoperability.

Quantum Simulation API Reference

Execute the current circuit DSL over HTTP and inspect exact state probabilities, seeded measurements and generated diagrams.

Documentation version: 2.0 · Compatible with: current AutomationGlance simulator · Last reviewed: 2026-10-05

[Open API playground](#) · [Full DSL specification](#) →

Endpoint and host

POST /api/runSimulation with Content-Type: application/json. On the live site, frontend/config/settings.js sets the API base to https://quantum-playground-flex-gag3cjd5aedvf6dz.southindia-01.azurewebsites.net. Append the endpoint to that host; the site's https://automationglance.com/api/ serves this documentation. Azure Functions local development normally uses port 7071; the repository integration fixture uses 7073.

Request fields

Simulation request object

Field	Type / default	Contract
circuit	string, required	Nonempty supported DSL; configured default string limit 10,000 characters
shots	integer, 1024	1–100,000 inclusive
seed	integer or null, omitted	Signed 32-bit (–2147483648...2147483647); omitted/null derives a stable seed
sessionId	string or null	Required and nonblank when monetization is enabled; use the session identifier associated with your usage allocation. Ignored for quota purposes when disabled.

Field names are case-insensitive. Unknown fields, malformed JSON, comments and trailing commas are rejected by payload validation. Send each field once. No noise, backend, format, or gate-definition option is supported by this endpoint. No API key is required by this controller; deployment infrastructure may add restrictions.

```
{
  "circuit": "qubits 2\nH 0\nCX 0 1\nMEASURE 0\nMEASURE 1",
  "shots": 1024,
  "seed": 42,
  "sessionId": "documentation-local-session"
}
```

The example session is for the local fixture. Use your own existing session for a monetized deployment; examples do not bypass usage limits. Invalid DSL is parsed after the simulator quota guard, so an unsuccessful DSL execution can consume a run.

Successful response (HTTP 200)

Public JSON response fields

Field	Type and meaning
success	boolean, true on success
probabilities	array of {state: string, probability: number}; full computational-basis distribution, independent of shots
counts	object from measured bitstring to integer sampled count; only sampled keys appear; empty if no measurements
blochVectors	array of {x,y,z} numbers, one reduced single-qubit Bloch vector per qubit
stateVector	array of {state: string, real: number, imag: number}; full final pre-measurement amplitudes in increasing basis index
metadata	{seed: integer, shots: integer, numQubits: integer, backendVersion: string}
errors	array of {line: integer or null, message: string, type: string, code: string or null}; successful parses can contain warnings
circuit_image_base64	string or null; raw base64 SVG payload
circuitDiagram	string or null; equivalent SVG data URL, ready for image display

For the Bell request, probabilities are 0.5 for 00 and 11 and zero for 01 and 10, up to floating-point roundoff. Each qubit's Bloch vector is (0,0,0). Counts sum to shots but need not split equally. Do not interpret the statevector as the result of physical hardware execution.

Full-state labels put q0 leftmost. Counts follow MEASURE statement order and may have a different width. All gates run before sampling; no mid-circuit feedback. See [measurement ordering](#).

HTTP errors

Implemented controller and middleware statuses

Status	When
400	Invalid JSON/shape/unknown fields, type deserialization failure, string too long, missing required monetized session
402	Free and paid simulation runs exhausted when monetization is enabled
413	Request body exceeds configured byte limit
422	Missing/blank circuit, invalid shots, invalid DSL, width >8, or simulation validation failure
429	Enabled per-client rate limit exceeded; Retry-After indicates wait seconds
500	Unexpected parsing or simulation failure

Error objects include `error` and `correlationId`; `detail` or `message` may be present. DSL failure includes `parse_errors`; do not assume the success response's `errors` field. A width error also supplies `requested` and `limit`. A quota error supplies `paymentEndpoint`. Preserve the correlation ID when diagnosing failures.

Limits and reproducibility

Execution: 1–8 qubits and 1–100,000 shots. Checked-in security defaults: 1,048,576 request bytes, 10,000 characters per string, and 60 requests per 60-second window per client (normally IP). These security settings are deployment-configurable; they are not a guarantee of production quota. This controller has no separate gate-count limit; circuit text is bounded by request limits. Session usage limits are separate from rate limits.

Explicit seed controls only sampling. Otherwise 32-bit FNV-1a hashes UTF-8 bytes of the exact `circuit + "|" + shots` string. Editing whitespace or shots changes the derived seed. Metadata returns the seed and backend version; reproducibility assumes the same circuit, shots, seed and backend/runtime. Probabilities and amplitudes are deterministic ideal calculations independent of seed.

Executable client examples

These examples use the local fixture by default. Set `AUTOMATIONGLANCE_API_BASE` to your API host and `AUTOMATIONGLANCE_SESSION_ID` to your existing session for another deployment. Each checks HTTP failure before using results.

curl (bash and Python 3 for JSON encoding)

[Download curl \(bash and Python 3 for JSON encoding\) example](#)

```
#!/usr/bin/env bash
set -euo pipefail
# Point to your API host; the local integration fixture listens on port 7073.
: "${AUTOMATIONGLANCE_API_BASE:=http://127.0.0.1:7073}"
: "${AUTOMATIONGLANCE_SESSION_ID:=documentation-local-session}"
# Generate JSON safely; never interpolate arbitrary input into a JSON string.
python3 - <<'JSON' | curl --fail-with-body -sS
"$AUTOMATIONGLANCE_API_BASE/api/runSimulation" -H 'Content-Type: application/json' -
-data-binary @-
import json, os
print(json.dumps({"circuit": "qubits 2\nH 0\nCX 0 1\nMEASURE 0\nMEASURE 1", "shots":
1024, "seed": 42,
                  "sessionId": os.environ.get("AUTOMATIONGLANCE_SESSION_ID",
"documentation-local-session"))))
JSON
```

C# (.NET 10)

[Download C# \(.NET 10\) example](#)

```
#:property PublishAot=false
// .NET 10 file-based app: dotnet run --file csharp.cs
using System.Net.Http.Json;
using System.Text.Json;
var baseUrl = Environment.GetEnvironmentVariable("AUTOMATIONGLANCE_API_BASE") ??
"http://127.0.0.1:7073";
var payload = new {
    circuit = "qubits 2\nH 0\nCX 0 1\nMEASURE 0\nMEASURE 1", shots = 1024, seed =
42,
    sessionId = Environment.GetEnvironmentVariable("AUTOMATIONGLANCE_SESSION_ID") ??
"documentation-local-session"
};
using var client = new HttpClient();
using var response = await client.PostAsJsonAsync(baseUrl + "/api/runSimulation",
payload);
var body = await response.Content.ReadAsStringAsync();
if (!response.IsSuccessStatusCode) throw new HttpRequestException($"
{(int)response.StatusCode}: {body}");
using var result = JsonDocument.Parse(body);
Console.WriteLine(result.RootElement.GetRawText());
```

JavaScript (Node.js 18+)

[Download JavaScript \(Node.js 18+\) example](#)

```
// Node.js 18+; for browser code use window.API_BASE and your existing session ID.
const base = process.env.AUTOMATIONGLANCE_API_BASE || 'http://127.0.0.1:7073';
const payload = {
  circuit: 'qubits 2\nH 0\nCX 0 1\nMEASURE 0\nMEASURE 1', shots: 1024, seed: 42,
  sessionId: process.env.AUTOMATIONGLANCE_SESSION_ID || 'documentation-local-session'
};
const response = await fetch(`${base}/api/runSimulation`, {
  method: 'POST', headers: { 'Content-Type': 'application/json' }, body:
JSON.stringify(payload)
});
if (!response.ok) throw new Error(`${response.status}: ${await response.text()}`);
console.log(JSON.stringify(await response.json()));
```

Python 3

Download Python 3 example

```
# Python 3 standard library.
import json, os, urllib.request, urllib.error
base = os.environ.get("AUTOMATIONGLANCE_API_BASE", "http://127.0.0.1:7073")
payload = {"circuit": "qubits 2\nH 0\nCX 0 1\nMEASURE 0\nMEASURE 1", "shots": 1024,
"seed": 42,
"sessionId": os.environ.get("AUTOMATIONGLANCE_SESSION_ID",
"documentation-local-session")}
request = urllib.request.Request(base + "/api/runSimulation",
data=json.dumps(payload).encode(),
headers={"Content-Type": "application/json"},
method="POST")
try:
  with urllib.request.urlopen(request) as response:
    print(json.dumps(json.load(response)))
except urllib.error.HTTPError as error:
  raise RuntimeError(f"{error.code}: {error.read().decode()}") from error
```

Quantum Modeler endpoints and exports

POST /api/v1/quantum-model/maxcut, /api/v1/quantum-model/knapsack and /api/v1/quantum-model/qubo return executed, verified and projected results separately. Use the corresponding [modeler pages](#); Custom QUBO also provides Copy API JSON.

Modeler downloads: DSL and OpenQASM 2.0; Custom QUBO additionally exports automationglance.qubo.v1 JSON with additive objective, variable-order and pair-convention metadata. Keep only coefficient fields when constructing a strict execution request. Simulator image downloads provide SVG and PNG after rendering a current circuit.

AutomationGlance displays $x_0 \dots x_{N-1}$; modeler circuits map $x_i \rightarrow q[i] \rightarrow c[i]$. Some tools display classical registers in reverse order $c[N-1] \dots c[0]$. Use the measurement mapping before comparing counts. This documents conventions, not tested external-QPU or SDK interoperability.

Optional signed 32-bit seed makes sampling reproducible; omission retains the simulator's circuit/shot-derived default. Modeler handoff preserves its explicit seed.

Where AutomationGlance Fits

A lightweight browser-first environment for learning, formulation, bounded experimentation, classical cross-checking and export.

Documentation version: 2.0 · Compatible with: current AutomationGlance simulator · Last reviewed: 2026-10-05

AutomationGlance is not intended to replace these ecosystems. Its 8-qubit execution limit is a classical simulator boundary, not a representation of modern hardware scale. Exported circuits are starting points for further experimentation.

Purpose, not a capability ranking

Tool	Primary purpose
AutomationGlance	Learning, bounded circuit simulation, optimization formulation and classical validation
Qiskit	Quantum SDK and tooling within the IBM Quantum ecosystem
Cirq	Python framework for writing, manipulating and optimizing quantum circuits
PennyLane	Hybrid quantum/classical computation and differentiable quantum programming
D-Wave Ocean	Tools for formulating problems, including QUBO/Ising models, in the D-Wave annealing and hybrid-solver ecosystem

Descriptions are based on the linked official project documentation, reviewed 2026-10-05. These projects have broader capabilities than this short overview. No comparative performance or interoperability guarantee is implied.

Quantum Computing References

Primary papers, specifications, official project documentation, and implementation entry points.

Documentation version: 2.0 · Compatible with: current AutomationGlance simulator · Last reviewed: 2026-10-05

- Farhi, Goldstone & Gutmann (2014), [A Quantum Approximate Optimization Algorithm](#). The implemented fixed $p=1$ grid is described separately in our methodology.
- Lucas (2014), [Ising formulations of many NP problems](#). Context for binary penalties and Ising formulations; our exact coefficient convention is documented in [the mapping](#).
- Cross et al. (2017), [Open Quantum Assembly Language; current OpenQASM specification](#). AutomationGlance exports a 2.0 subset, not all current-language features.
- Preskill (2018), [Quantum Computing in the NISQ era and beyond](#). NISQ refers to noisy intermediate-scale quantum devices; this ideal simulator does not model their noise.
- [IBM Quantum Learning: quantum circuits](#) and [Statevector reference](#) for the circuit model and statevector representation.
- The Max-Cut objective here is the unweighted sum of edges crossing a binary partition. The QAOA paper above treats this objective. See our [enumerated example](#).
- Official ecosystem references are linked from [Where AutomationGlance Fits](#).

Implementation entry points

The repository is the source for product-specific behavior. At review baseline

3fb7ce24194cf19addcf387fc80a9674f1f9582, inspect [src/QuantumPlayground](#):

CircuitParserService, QuantumSimulatorService, SimulationController, BinaryQuadraticModel,

QaoaCircuitBuilderService, QaoaParameterOptimizerService, QuboQaoaService, the three

classical solvers and scale estimators. Browser export lives in frontend/assets/qasm.js. Repository

regression tests accompany this documentation update.