

Automated Heuristic Optimisation for Encoded Magic-State Preparation on the $[[4, 2, 2]]$ Code

A Karpathy-Style Autoresearch Ratchet
for Quantum Error Detection Experiments

System design and implementation by Claude (Anthropic)
Research direction and project ownership by the author

<https://github.com/saymrwulf/autoresearch-quantum>

April 2026

Abstract

We present AUTORESEARCH-QUANTUM, an automated experimental optimisation system for encoded magic-state preparation on the $[[4, 2, 2]]$ quantum error-detecting code. The system implements a five-rung progressive search ratchet—inspired by Karpathy’s autoresearch pattern—that discovers high-quality preparation recipes through simulated noisy evaluation, statistical lesson extraction, and cross-rung knowledge propagation. Starting from a combinatorial space of $\sim 10^4$ possible experiment configurations (seed gates, encoder circuits, verification protocols, transpilation strategies), the ratchet autonomously narrows to a near-optimal recipe without human intervention between rungs. We describe the quantum-mechanical observables being optimised, the search and scoring machinery, the machine-readable feedback loop that closes the “auto” in autoresearch, and a complete test suite that anchors each architectural claim to an executable proof. The system runs entirely on classical simulation via Qiskit Aer and is designed for eventual promotion to IBM Quantum hardware.

Contents

1	Introduction and Motivation	3
2	The Quantum Problem	3
2.1	The $[[4, 2, 2]]$ Error-Detecting Code	3
2.2	Magic States and the T Gate	4
2.3	Preparation Circuit Architecture	4
2.4	Verification and Postselection	5
2.5	Measurement Protocol and Observables	6
2.6	The Magic-State Witness	6
3	The Scoring Functions	7

3.1	Weighted Acceptance-Cost Score (Rungs 1–3)	7
3.2	Factory Throughput Score (Rungs 4–5)	7
4	The Autoresearch Engine	8
4.1	The Ratchet Mechanism	8
4.2	Multi-Rung Propagation	9
4.3	Challenger Generation Strategies	10
4.3.1	NeighborWalk (40% budget)	10
4.3.2	RandomCombo (30% budget, or 60% when no lessons exist)	10
4.3.3	LessonGuided (30% budget, only when lessons exist)	11
5	The Lesson Feedback System	11
5.1	Rule Extraction	11
5.2	Search Space Narrowing	12
5.3	Dual Output Format	12
6	Transfer Evaluation	13
7	The Five Rungs in Detail	13
8	Resumability and Persistence	14
9	Verification: Claims and Tests	14
9.1	Quantum Correctness (3 tests)	14
9.2	Search Correctness (5 tests)	15
9.3	Lesson Correctness (3 tests)	15
9.4	Scoring Correctness (2 tests)	16
9.5	Transfer Correctness (1 test)	16
9.6	Persistence Correctness (2 tests)	16
9.7	Integration (4 tests)	16
10	Usage Without AI Assistance	17
10.1	Installation	17
10.2	CLI Reference	17
10.3	The Feedback Artefacts and How to Read Them	17
10.4	Manual Iteration Workflow	18
11	Limitations and Future Work	18
12	Conclusion	19

1 Introduction and Motivation

Fault-tolerant quantum computation requires non-Clifford gates. The standard approach is to prepare *magic states*—resource states that, when consumed by gate teleportation, implement a T gate ($\pi/8$ rotation) on a logical qubit [1]. The quality of these raw magic states directly governs the cost of downstream distillation: higher input fidelity means fewer distillation rounds, which means fewer physical qubits and less wall-clock time.

Preparing an encoded magic state is not a single circuit—it is a family of choices: how to seed the initial rotation, how to encode into the code’s logical subspace, whether to verify stabilisers, how aggressively to postselect, how to transpile for a specific device topology. Each choice interacts with the others in ways that are difficult to predict analytically, especially under realistic device noise.

Key Concept

The core question is combinatorial, not algebraic: *which combination of preparation, verification, and transpilation choices yields the highest-quality encoded magic states at the lowest cost, under a given noise model?* This is an experimental optimisation problem, and it is the kind of problem that machines are better at than humans—provided you give the machine a well-defined objective and a mechanism to learn from its own experiments.

AUTORESEARCH-QUANTUM is that mechanism. It is a direct implementation of the *autoresearch pattern* described by Karpathy [2]: an automated loop that proposes experiments, evaluates them, extracts lessons, and uses those lessons to propose better experiments. The system produces two outputs: an optimised experiment recipe, and a structured explanation of why that recipe works.

2 The Quantum Problem

2.1 The $[[4, 2, 2]]$ Error-Detecting Code

The $[[4, 2, 2]]$ code is the smallest quantum error-detecting code. It encodes $k = 2$ logical qubits into $n = 4$ physical qubits and has distance $d = 2$: it can detect any single-qubit error, but cannot correct it. Its stabiliser group is generated by two operators:

$$S_X = X^{\otimes 4} = XXXX, \quad S_Z = Z^{\otimes 4} = ZZZZ. \quad (1)$$

Any state $|\psi\rangle$ in the code space satisfies $S_X |\psi\rangle = S_Z |\psi\rangle = +|\psi\rangle$. A single-qubit error E anticommutes with at least one stabiliser, so measuring the stabilisers reveals whether an error occurred without collapsing the logical information.

Intuition

Think of the stabilisers as parity checks. $S_Z = ZZZZ$ checks whether the total Z -parity of all four qubits is even. If a bit-flip error X_i hits one qubit, the parity flips from even to odd, and you detect it. $S_X = XXXX$ does the same in the X basis, catching phase-flip errors. Together, they detect any single error of any type—but they cannot tell you *which* qubit was affected, so correction is impossible. The response is to *discard* the shot (postselection).

The code has two logical qubits with the following representative logical operators:

$$\bar{X}_1 = X_1X_3, \quad \bar{Z}_1 = Z_1Z_2, \quad \bar{X}_2 = X_2X_3, \quad \bar{Z}_2 = Z_2Z_3. \quad (2)$$

We use logical qubit 1 to carry the magic state and logical qubit 2 as a *spectator*: an idle qubit whose \bar{Z}_2 measurement monitors whether the encoding process preserved the code space.

2.2 Magic States and the T Gate

The T gate applies the rotation $T = \text{diag}(1, e^{i\pi/4})$ to a single qubit. It cannot be implemented transversally on most stabiliser codes, so the standard workaround is *gate teleportation*: consume a pre-prepared resource state to apply T to a data qubit via Clifford operations and measurement.

The canonical magic state for the T gate is:

$$|T\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{i\pi/4}|1\rangle) = TH|0\rangle. \quad (3)$$

Our system prepares an *encoded* version: the magic state is placed on logical qubit 1 of the $[[4, 2, 2]]$ code, so that error detection protects it during storage and transport.

Subtlety

The $[[4, 2, 2]]$ code is too small for real fault tolerance—its distance of 2 means a two-qubit error goes undetected. We use it because it is the *simplest non-trivial testbed* for the engineering question of how to prepare, verify, and evaluate an encoded magic state. The heuristics discovered here—which seed gates help, whether X -stabiliser checking is worth the cost, how transpilation interacts with noise—are intended to transfer to larger codes.

2.3 Preparation Circuit Architecture

The preparation circuit has two stages:

Stage 1: Seed gate. A single-qubit rotation on physical qubit 0 creates the raw magic state before encoding. The system explores three decompositions, all mathematically equivalent on an ideal device but differing in their interaction with noise:

Name	Gate sequence	Rationale
<code>h_p</code>	H then $P(\pi/4)$	Native Qiskit decomposition
<code>ry_rz</code>	$R_y(\pi/2)$ then $R_z(\pi/4)$	Avoids the H gate entirely
<code>u_magic</code>	$U(\pi/2, 0, \pi/4)$	Single physical gate; minimal depth

All three produce the same ideal state $|T\rangle$. Under noise, they differ because each gate decomposition couples differently to coherent and incoherent error channels of the physical device.

Stage 2: Encoder. An entangling circuit maps the single-qubit state on qubit 0 into the four-qubit code space. Two implementations exist:

- **cx_chain:** A sequence of CNOT gates that propagates the logical information:

$$\text{CNOT}_{0 \rightarrow 2}, \quad \text{CNOT}_{1 \rightarrow 0}, \quad H_3, \quad \text{CNOT}_{3 \rightarrow 0}, \quad \text{CNOT}_{3 \rightarrow 1}, \quad \text{CNOT}_{3 \rightarrow 2}.$$

- **cz_compiled:** The same logical operation decomposed into CZ and Hadamard gates, which may be more natural for devices with a native CZ interaction.

2.4 Verification and Postselection

After encoding, the system optionally measures one or both stabilisers using ancilla qubits. This is *verification*: a check that the state is in the code space before proceeding to the logical measurement.

Definition 2.1 (Postselection). *Given N total shots, let $A \subseteq \{1, \dots, N\}$ be the set of shots where all measured syndromes are $+1$ (no error detected). The postselected ensemble consists of only the shots in A . The acceptance rate is $|A|/N$.*

The system supports four postselection modes:

Mode	Behaviour
<code>all_measured</code>	Discard a shot if <i>any</i> measured syndrome is -1
<code>z_only</code>	Discard only if the S_Z syndrome is -1
<code>x_only</code>	Discard only if the S_X syndrome is -1
<code>none</code>	Keep all shots regardless of syndrome

Intuition

Postselection is a quality–throughput trade-off. Aggressive postselection (`all_measured`) gives higher-quality surviving shots but discards more data, lowering the acceptance rate. The optimiser’s job is to find the mode that maximises the product of quality and throughput under the prevailing noise.

Two ancilla strategies exist:

- **dedicated_pair:** Each stabiliser gets its own ancilla qubit (two ancillas total for `both` verification). More gates, but no risk of mid-circuit reset errors.
- **reused_single:** One ancilla, measured and reset between stabiliser checks. Fewer qubits, but the reset introduces a potential error source.

2.5 Measurement Protocol and Observables

After preparation and optional verification, the system constructs four circuits that measure different observables on the same encoded state:

1. **Acceptance circuit.** Measures all data qubits in the Z basis. No basis rotation before measurement. Used to compute the acceptance rate.
2. **Logical X circuit.** Applies H to qubits $\{0, 2\}$ (the support of \bar{X}_1) before measurement, effectively measuring in the X basis on those qubits. The expectation value $\langle \bar{X}_1 \rangle$ is computed from the parity of the postselected outcomes on the relevant qubits.
3. **Logical Y circuit.** Applies S^\dagger then H to qubit 0 (Y -basis), Z -basis on qubit 1, and H on qubit 2 (X -basis). This measures the three-body operator $Y_0 Z_1 X_2$, which is the logical Y on qubit 1 of the $[[4, 2, 2]]$ code. The expectation $\langle \bar{Y}_1 \rangle$ is extracted from the postselected parity.
4. **Spectator Z circuit.** Applies H to neither qubit 1 nor qubit 2 (they are already in the Z basis). Measures the two-body operator $Z_1 Z_2 = \bar{Z}_2$, the logical Z of the spectator qubit.

Key Concept

A perfect encoded magic state satisfies $\langle \bar{X}_1 \rangle = \langle \bar{Y}_1 \rangle = 1/\sqrt{2}$ and $\langle \bar{Z}_2 \rangle = +1$. Any deviation indicates either a preparation error (the seed or encoder is wrong), a noise-induced error (the device corrupted the state), or a code-space leakage (the spectator was disturbed).

2.6 The Magic-State Witness

We combine the three observables into a single quality metric called the *logical magic witness*:

$$W = \underbrace{\frac{1 + \frac{\langle \bar{X}_1 \rangle + \langle \bar{Y}_1 \rangle}{\sqrt{2}}}{2}}_{\text{magic quality}} \cdot \underbrace{\frac{1 + \langle \bar{Z}_2 \rangle}{2}}_{\text{spectator alignment}}. \quad (4)$$

The first factor measures how close the logical qubit is to the ideal magic state. For $|T\rangle$, the Bloch vector has equal X and Y components of magnitude $1/\sqrt{2}$, so the numerator reaches its maximum of $1 + 1 = 2$ and the factor equals 1.

The second factor penalises any disturbance of the spectator qubit. An ideal encoding leaves the spectator in $|+\rangle_L$, giving $\langle \bar{Z}_2 \rangle = +1$ and a factor of 1. If the encoding process or noise leaks information into the spectator, $\langle \bar{Z}_2 \rangle$ drops, and so does W .

The witness is bounded: $W \in [0, 1]$, with $W = 1$ for the ideal encoded magic state.

Subtlety

W is not a fidelity. It is a *witness* in the sense that $W < 1$ proves the state is not ideal, but $W = 1$ does not uniquely identify the state (other states could also achieve $W = 1$ if the measured observables happen to align). For the purpose of heuristic optimisation, this is sufficient: we are maximising W as a proxy for magic-state quality, not proving fault tolerance.

3 The Scoring Functions

Raw metrics (witness, acceptance rate, gate count, depth) must be collapsed into a single scalar for the ratchet to compare experiments. The system provides two scoring functions, used at different stages of the search.

3.1 Weighted Acceptance-Cost Score (Rungs 1–3)

$$\text{score}_{\text{WAC}} = \frac{Q \cdot r_{\text{accept}}}{C} \quad (5)$$

where:

$$Q = \frac{\sum_i w_i \cdot q_i}{\sum_i w_i}, \quad q_i \in \{F_{\text{ideal}}, F_{\text{noisy}}, W, r_{\text{code}}, S, A_{\text{spec}}\}, \quad (6)$$

$$C = C_0 + \alpha_{2q} \cdot n_{2q} + \alpha_d \cdot d + \alpha_s \cdot n_s + \alpha_r \cdot t_r + \alpha_q \cdot c_q. \quad (7)$$

Symbol	Name	Definition
F_{ideal}	Ideal fidelity	$\langle T_{\text{enc}} \rho_{\text{ideal}} T_{\text{enc}} \rangle$, from statevector sim
F_{noisy}	Noisy fidelity	$\langle T_{\text{enc}} \rho_{\text{noisy}} T_{\text{enc}} \rangle$, from density-matrix sim with noise model
W	Witness	Eq. (4)
r_{code}	Codespace rate	Mean acceptance across all four circuits
S	Stability score	$1 - \sigma(\{s_j\})/\bar{s}$, where s_j is the score of repeat j
A_{spec}	Spectator alignment	$(1 + \langle \bar{Z}_2 \rangle)/2$
r_{accept}	Acceptance rate	Fraction of shots surviving postselection
n_{2q}	Two-qubit gates	Sum across all transpiled circuits
d	Circuit depth	Maximum across all transpiled circuits
n_s	Shot count	shots \times repeats \times circuits
t_r	Runtime estimate	Estimated wall-clock time
c_q	Queue cost proxy	0 for simulation; 1 for hardware

The weights w_i and cost coefficients α_* are per-rung hyperparameters defined in the YAML configuration. For example, Rung 1 puts 40% weight on noisy fidelity and 25% on the witness, while Rung 2 shifts 20% weight to stability.

Intuition

The score has a clear physical interpretation: it is *useful quality per unit cost*. The numerator $Q \cdot r_{\text{accept}}$ measures how much high-quality output you get. The denominator C measures what you pay in gates, depth, and shots. A circuit that achieves $W = 0.9$ but requires 80 two-qubit gates is penalised relative to one that achieves $W = 0.85$ with only 30 gates.

3.2 Factory Throughput Score (Rungs 4–5)

For the later rungs, the question shifts from “what is the best state?” to “what gives the most accepted states per unit cost?”—a factory-style metric:

$$\text{score}_{\text{factory}} = \frac{r_{\text{accept}} \cdot W}{C'} \quad (8)$$

where C' uses a $1.5\times$ heavier penalty on two-qubit gate count and circuit depth than C . This is not a theoretical derivation; it is an engineering choice that biases the optimiser toward cheaper circuits when the quality is already good enough.

In addition to the scalar score, the factory scorer computes and stores auxiliary metrics:

$$\text{throughput} = \frac{r_{\text{accept}}}{C'}, \quad \text{logical error} = 1 - W, \quad \text{cost per accepted} = \frac{C'}{r_{\text{accept}}}. \quad (9)$$

These are attached to the experiment record for downstream analysis.

4 The Autoresearch Engine

4.1 The Ratchet Mechanism

The ratchet is a monotone optimisation procedure: the incumbent (best-known experiment) can only improve or stay the same, never regress.

Algorithm 1: Run-Rung: a complete search campaign

Input : Rung config \mathcal{R} , step budget B , patience P **Output**: Steps \mathcal{S} , lesson ℓ , feedback ϕ

```
1  $x^* \leftarrow \text{ENSUREINCUMBENT}(\mathcal{R});$ 
2  $p \leftarrow P;$ 
3 for  $t \leftarrow 1$  to  $B$  do
4    $\mathcal{G} \leftarrow \text{COMPOSITEGENERATOR}(x^*, \text{dims}(\mathcal{R}), H, \Phi);$ 
5   foreach challenger  $c \in \mathcal{G}$  do
6     Evaluate  $c$  on cheap tier;
7      $H \leftarrow H \cup \{\text{fingerprint}(c)\};$ 
8   end
9    $\hat{c} \leftarrow \arg \max_{c \in \mathcal{G}} \text{score}(c);$ 
10  if  $\text{score}(\hat{c}) > \text{score}(x^*) + \epsilon$  then
11     $x^* \leftarrow \hat{c};$ 
12     $p \leftarrow P;$                                      // Reset patience
13  else
14     $p \leftarrow p - 1;$ 
15  end
16  Save progress checkpoint;
17  if  $p \leq 0$  then break ;                               // Early stopping
18  ;
19 end
20  $(\ell, \phi) \leftarrow \text{EXTRACTLESSON}(\text{all experiments in } \mathcal{R});$ 
21  $\Phi \leftarrow \Phi \cup \{\phi\};$ 
22 return  $\text{steps}, \ell, \phi;$ 
```

Key design choices:

- **History deduplication** (H): the set of all fingerprints ever evaluated, across all steps of the current rung. No experiment is run twice.
- **Patience** (p): if the incumbent survives P consecutive steps without being dethroned, the rung terminates early. This prevents wasting budget when the search has converged.
- **Promotion threshold** (ϵ): a challenger must beat the incumbent by at least ϵ (the `cheap_margin`) to be considered for promotion. This prevents churn from noise-band fluctuations.

4.2 Multi-Rung Propagation

When the ratchet runs multiple rungs in sequence, it propagates knowledge forward:

Algorithm 2: Run-Ratchet: progressive multi-rung search

Input : Rung configs $[\mathcal{R}_1, \dots, \mathcal{R}_K]$

```
1  $\Phi \leftarrow \emptyset$ ; // Accumulated lesson feedback
2 for  $i \leftarrow 1$  to  $K$  do
3   if  $i > 1$  then
4     Override  $\mathcal{R}_i$ .bootstrap with winner spec from  $\phi_{i-1}$ ;
5      $\mathcal{R}_i$ .search_space  $\leftarrow$  NARROW( $\mathcal{R}_i$ .search_space,  $\Phi$ );
6   end
7    $(\_, \ell_i, \phi_i) \leftarrow \text{RUNRUNG}(\mathcal{R}_i, \Phi)$ ;
8    $\Phi \leftarrow \Phi \cup \{\phi_i\}$ ;
9 end
```

Two things happen between rungs:

1. **Winner propagation.** The best experiment spec from rung i becomes the starting point (bootstrap incumbent) for rung $i+1$. The human-written YAML bootstrap is overridden. A `propagated_spec.json` is saved for traceability.
2. **Search space narrowing.** The accumulated `SearchRule` objects from all completed rungs are combined and used to prune the dimension grid:
 - Dimensions with a `fix` rule are constrained to a single value.
 - Values with an `avoid` rule are removed, unless doing so would reduce a dimension below 2 values.

Intuition

Rung 1 searches broadly: 3 seed styles \times 2 encoder styles \times 3 verification modes \times 3 postselection modes $\times \dots \approx 10^3$ combinations. By Rung 5, lessons have fixed the seed style, fixed the encoder, and eliminated half the verification modes. The search space might be down to ~ 20 combinations. The later rungs are not wasting budget re-exploring dead ends.

4.3 Challenger Generation Strategies

The system uses a composite generator that allocates its per-step budget across three strategies:

4.3.1 NeighborWalk (40% budget)

The simplest strategy: for each dimension in the search space, try every alternative value while holding all other dimensions fixed. This is a single-axis perturbation, equivalent to one step of coordinate descent. It is thorough within a Hamming distance of 1 from the incumbent but cannot escape local optima.

4.3.2 RandomCombo (30% budget, or 60% when no lessons exist)

Picks 1–3 dimensions uniformly at random and samples a new value for each. This produces multi-axis mutations that can jump across the landscape:

```

n_dims = random.randint(1, min(3, len(dim_names)))
chosen_dims = random.sample(dim_names, n_dims)
for dim in chosen_dims:
    alternatives = [v for v in values if v != current]
    updates[dim] = random.choice(alternatives)

```

The key property: unlike NeighborWalk, this can change **verification** and **postselection** simultaneously, which matters because these two dimensions interact (see §5.1).

4.3.3 LessonGuided (30% budget, only when lessons exist)

Reads the **SearchRule** list from accumulated lesson feedback and constructs candidates that respect the rules:

1. Apply all **fix** rules unconditionally (e.g., always set **seed_style=ry_rz**).
2. For dimensions with **prefer** rules, sample from preferred values with probability proportional to the rule’s confidence score.
3. For dimensions with **avoid** rules, exclude those values from sampling.
4. With 50% probability, also explore non-preferred, non-avoided values to prevent premature convergence.

All three strategies check every generated candidate against the global history set and discard duplicates before returning.

5 The Lesson Feedback System

The feedback system is the component that makes the ratchet *intelligent* rather than merely *persistent*. It transforms raw experiment data into actionable directives.

5.1 Rule Extraction

Given N experiment records from a completed rung, the system extracts three types of rules:

Per-dimension mean effects. For each dimension d and each value v in the search space, compute:

$$\delta_{d,v} = \bar{s}_{d=v} - \bar{s}_{\text{overall}} \quad (10)$$

where $\bar{s}_{d=v}$ is the mean score of all experiments that used value v for dimension d . If $\delta > \tau$ (default threshold $\tau = 0.005$), emit a **prefer** rule. If $\delta < -\tau$, emit an **avoid** rule. The confidence is $\min(1, n_{d=v}/N)$, where $n_{d=v}$ is the number of experiments with that value.

Fix detection. Let \mathcal{T}_K be the top- K experiments by score, where $K = \min(5, \max(3, \lfloor N/3 \rfloor))$. For each dimension, if all experiments in \mathcal{T}_K share the same value v^* , and $\bar{s}_{d=v^*} > \bar{s}_{d \neq v^*}$, emit a **fix** rule with confidence K/N .

Interaction detection. For each pair of dimensions (d_a, d_b) , compute the joint effect and compare it to the sum of marginals:

$$I_{v_a, v_b} = \underbrace{(\bar{s}_{d_a=v_a, d_b=v_b} - \bar{s}_{\text{overall}})}_{\text{joint effect}} - \underbrace{(\delta_{d_a, v_a} + \delta_{d_b, v_b})}_{\text{sum of marginals}}. \quad (11)$$

If $|I| > 2\tau$, the pair exhibits a synergy ($I > 0$) or conflict ($I < 0$) that the marginal analysis would miss. A rule is emitted for the composite dimension $d_a + d_b$.

Key Concept

Interaction detection matters because the $\llbracket 4, 2, 2 \rrbracket$ experiment has genuine dimension couplings. For example, `verification=z_only` combined with `postselection=z_only` may outperform the additive prediction because Z -only postselection is exactly matched to Z -only verification: no shots are wasted on X -syndrome failures that cannot be postselected against.

5.2 Search Space Narrowing

The `narrow_search_space()` function applies the extracted rules to physically shrink the dimension grid:

```
for dim, values in search_space.dimensions.items():
    if dim in fix_map and fix_map[dim] in values:
        new_dims[dim] = [fix_map[dim]]          # Lock to one value
    elif dim in avoid_map:
        filtered = [v for v in values if v not in avoid_map[dim]]
        if len(filtered) >= min_values_per_dim:
            new_dims[dim] = filtered              # Remove bad values
        else:
            new_dims[dim] = list(values)         # Safety: keep all
```

The safety clause prevents over-pruning: if avoiding too many values would leave a dimension with fewer than 2 options, the original set is kept. Confidence thresholds (≥ 0.3 for avoid, ≥ 0.4 for fix) provide additional conservatism.

5.3 Dual Output Format

Every rung produces two artefacts:

1. **RungLesson** (human-readable Markdown): sections for “What Helped”, “What Hurt”, “Invariants”, “Hardware-Specific Effects”, “Next Tests”, “Promote Forward”, and “Discard”. Saved as `lesson.md`.
2. **LessonFeedback** (machine-readable JSON): the list of `SearchRule` objects, the narrowed dimension grid, the best spec’s field values, and optional transfer scores. Saved as `lesson_feedback.json` and consumed by the `LessonGuided` strategy and the propagation logic.

6 Transfer Evaluation

Rung 3 asks: does this recipe generalise, or is it overfitting to one backend’s noise profile? The `TransferEvaluator` answers this by running the *same* spec on multiple backend noise models and scoring pessimistically:

$$s_{\text{transfer}} = \min_{b \in \mathcal{B}} s_b(\text{spec}) \quad (12)$$

where \mathcal{B} is the set of backends (e.g., `fake_brisbane`, `fake_kyoto`, `fake_sherbrooke`) and s_b is the WAC score under backend b ’s noise model.

```
python -m autoresearch_quantum run-transfer \
  --config configs/runs/rung3.yaml \
  --backends fake_brisbane fake_kyoto fake_sherbrooke
```

Subtlety

This is different from searching over `target_backend` as a dimension (which the original Codex implementation did). Searching over backends finds the *easiest* backend. Transfer evaluation finds specs that work *everywhere*—a much harder and more useful criterion.

7 The Five Rungs in Detail

Rung	Name	Question	Score	Steps	Key shift
1	Baseline	What recipe works at all?	WAC	3	Broad search
2	Stability	Is it stable under noise variation?	WAC	3	+20% stability weight
3	Transfer	Does it work on other devices?	WAC	3	Pessimistic cross-backend
4	Factory	What maximises throughput/-cost?	Factory	3	1.5× cost penalty
5	Rosenfeld	Which heuristics are load-bearing?	Factory	4	Narrowest search space

Table 1: The five rungs and their progressive focus.

Each rung YAML configures different quality weights. Rung 1 puts 40% weight on noisy fidelity (because early on, we want to find specs that even *approach* the ideal under noise). By Rung 2, 20% shifts to stability. By Rung 5, the factory throughput score replaces WAC entirely, and stability carries 25% weight.

The bootstrap incumbent for Rung 1 is human-chosen. For Rungs 2–5, it is automatically overridden by the winner from the previous rung via propagation (§4.2).

8 Resumability and Persistence

All state is persisted as JSON files under a store directory:

```
data/default/  
  rung_1/  
    experiments/          # One JSON per evaluated spec  
      r1-incumbent-a3f2b8c901.json  
      r1-challenger-7e9d1f0ab3.json  
      ...  
    ratchet_steps/        # One JSON per step  
      step_0001.json  
    incumbent.json        # {"experiment_id": "r1-incumbent-..."}  
    progress.json         # Resumability checkpoint  
    lesson.json           # Machine-readable lesson  
    lesson.md             # Human-readable narrative  
    lesson_feedback.json  # SearchRules for next rung  
  rung_2/  
    propagated_spec.json  # Winner carried from rung 1  
    ...
```

The `progress.json` file records the exact state needed to resume:

```
@dataclass  
class RungProgress:  
    rung: int  
    steps_completed: int  
    patience_remaining: int  
    current_incumbent_id: str  
    completed: bool = False
```

If the process is interrupted, re-running the same command detects the incomplete progress and resumes from the last checkpoint. No experiment is re-evaluated, and the patience counter is preserved.

9 Verification: Claims and Tests

The test suite contains 21 tests, each anchored to a specific architectural claim. We present them grouped by subsystem, with the falsification condition for each.

9.1 Quantum Correctness (3 tests)

Claim 9.1. *The ideal encoded magic state is a $+1$ eigenstate of both stabilisers.*

Test: `test_encoded_target_state_satisfies_stabilizers`. Computes $\langle S_X \rangle$ and $\langle S_Z \rangle$ on the statevector produced by the default preparation circuit. Asserts $|\langle S \rangle - 1| < 10^{-8}$. *Would fail if:* the encoder circuit is wrong, or the seed gate prepares the wrong single-qubit state.

Claim 9.2. *The measurement bundle contains exactly the four expected circuits with correct metadata.*

Test: `test_circuit_bundle_contains_expected_contexts`. *Would fail if:* a circuit is missing, mislabelled, or lacks the `logical_operator` metadata needed by the analysis code.

Claim 9.3. *The local executor produces positive scores with physically meaningful metrics under noisy simulation.*

Test: `test_local_executor_produces_score`. Runs the full evaluation pipeline (build, transpile, simulate, postselect, score) and checks $s > 0$, $0 \leq r_{\text{accept}} \leq 1$, $0 \leq W \leq 1$. *Would fail if:* any stage of the pipeline is broken, e.g., the transpiler produces an invalid circuit, or the noise model is incompatible.

9.2 Search Correctness (5 tests)

Claim 9.4. *NeighborWalk mutates exactly one dimension per challenger.*

Test: `test_neighbor_challengers_mutate_single_dimension`. For each generated challenger, counts the number of fields that differ from the incumbent. Asserts the count is exactly 1.

Claim 9.5. *History deduplication prevents re-evaluation.*

Test: `test_neighbor_walk_respects_history`. Generates challengers, passes their fingerprints as history, regenerates. Second call must return an empty list.

Claim 9.6. *RandomCombo produces multi-axis mutations.*

Test: `test_random_combo_generates_multi_axis_mutations`. With 3 dimensions and 10 candidates, at least one must differ from the incumbent in > 1 field. (Probabilistic, but failure probability is $< 10^{-6}$.)

Claim 9.7. *LessonGuided respects fix rules.*

Test: `test_lesson_guided_uses_rules`. Given a `fix` rule for `seed_style=ry_rz`, every generated challenger must have `spec.seed_style == "ry_rz"`.

Claim 9.8. *The composite generator stays within the budget cap.*

Test: `test_composite_generator_combines_strategies`. Asserts $0 < |\mathcal{G}| \leq \text{max_challengers}$.

9.3 Lesson Correctness (3 tests)

Claim 9.9. *Per-dimension analysis emits correct prefer/avoid directives.*

Test: `test_extract_search_rules_prefer_and_avoid`. With synthetic data where `z_only` scores ~ 0.82 and both scores ~ 0.52 , the rules must contain (`verification`, `prefer`, `z_only`) and (`verification`, `avoid`, `both`).

Claim 9.10. *Search space narrowing removes avoided values and constrains fixed dimensions.*

Test: `test_narrow_search_space_removes_avoided`. After applying an `avoid` rule for `x_only` and a `fix` rule for `ry_rz`, asserts `x_only` is absent and the fixed dimension has exactly one value.

Claim 9.11. *The end-to-end feedback pipeline produces correct best-spec fields.*

Test: `test_build_lesson_feedback_end_to_end`. With synthetic data, the `best_spec_fields` must report the top-scoring experiment's spec.

9.4 Scoring Correctness (2 tests)

Claim 9.12. *The factory score computes throughput metrics and attaches them to the record.*

Test: `test_factory_throughput_score_produces_metrics`. With known inputs ($r_{\text{accept}} = 0.7$, $W = 0.8$), verifies $s > 0$ and `metrics.extra["factory_metrics"]["accepted_states_per_shot"] == 0.70`.

Claim 9.13. *Both scoring functions are registered and dispatchable by name.*

Test: `test_score_registry_has_factory`. Asserts "factory_throughput" is a key in `SCORE_REGISTRY`.

9.5 Transfer Correctness (1 test)

Claim 9.14. *The transfer evaluator produces a valid report with per-backend scores.*

Test: `test_transfer_evaluator_runs_across_backends`. Runs a single-backend transfer evaluation and checks that the report has the correct type and a positive transfer score.

9.6 Persistence Correctness (2 tests)

Claim 9.15. *Progress checkpoints survive serialisation round-trips.*

Test: `test_save_and_load_progress`. Writes a `RungProgress` to disk, reads it back, verifies all fields match.

Claim 9.16. *Lesson feedback (including SearchRule objects) survives serialisation.*

Test: `test_save_and_load_lesson_feedback`. Round-trips a `LessonFeedback` with one rule, verifies the rule's dimension and action are preserved.

9.7 Integration (4 tests)

Claim 9.17. *A full rung run saves progress and marks completion.*

Test: `test_run_rung_saves_progress`. After `run_rung()`, `progress.json` must exist with `completed=true`.

Claim 9.18. *A full rung returns both a human lesson and machine feedback.*

Test: `test_run_rung_returns_lesson_and_feedback`. Checks the return type is a 3-tuple of (steps, `RungLesson`, `LessonFeedback`).

Claim 9.19. *Multi-rung ratchet propagates winners and accumulates lessons.*

Test: `test_run_ratchet_propagates_winner`. Runs a two-rung ratchet and asserts `len(harness._accumulated) == 2`.

Claim 9.20. *Different specs receive different simulator seeds.*

Test: `test_different_specs_get_different_seeds`. Computes SHA-256(`fingerprint||repeat_index`) for two specs with different `verification` values. The seeds must differ. *Regression test:* the original code used $11000 + i$ for all specs.

10 Usage Without AI Assistance

10.1 Installation

```
cd autoresearch-quantum
python -m venv .venv && source .venv/bin/activate
pip install -e ".[dev]"
python -m pytest tests/ -v          # 21 tests, ~13 seconds
```

Requires Python ≥ 3.11 and Qiskit ≥ 2.3 . No GPU needed.

10.2 CLI Reference

Command	What it does
<code>run-experiment</code>	Evaluate a single spec. Use <code>-set key=val</code> to override fields.
<code>run-step</code>	One ratchet step: generate challengers, evaluate, maybe update incumbent.
<code>run-rung</code>	Full rung: repeated steps until budget or patience exhausted. Produces lesson.
<code>run-ratchet</code>	Multiple rungs in sequence with automatic propagation.
<code>run-transfer</code>	Evaluate one spec across multiple backends.

10.3 The Feedback Artefacts and How to Read Them

After running `run-rung -config configs/rungs/rung1.yaml`:

lesson.md is the human report. Skim the “What Helped” and “What Hurt” sections first—they tell you which knobs to turn. The “Invariants” section tells you which knobs don’t seem to matter (yet).

lesson_feedback.json is the machine report. The `rules` array is the most important part:

```
{
  "dimension": "verification",
  "action": "prefer",
  "value": "z_only",
  "confidence": 0.67,
  "reason": "mean score 0.1823 is +0.0312 above overall mean (8 samples)"
}
```

Read it as: “With 67% confidence (based on 8 out of 12 experiments), `verification=z_only` helps.” A confidence below 0.3 means the system saw too few samples to be sure.

incumbent.json points to the current best. Load the corresponding experiment file to see its full spec and all metrics.

propagated_spec.json (Rungs 2–5 only) is the spec that was carried forward. Compare it with the YAML bootstrap to see what changed automatically.

10.4 Manual Iteration Workflow

1. Run Rung 1. Read `lesson.md`.
2. Edit `rung2.yaml`: remove values that the lesson says to avoid. Add new values for dimensions you want to explore.
3. Run Rung 2. Or run the full ratchet and let propagation handle it.
4. Open `lesson_feedback.json` from Rung 1 and Rung 3 side by side. Rules that appear in both with high confidence are load-bearing. Rules that appear only early on were artefacts of a specific noise model.
5. Test specific hypotheses: `run-experiment -set approximation_degree=0.95`. The result joins the store and influences the next lesson extraction.
6. If interrupted, just re-run the same command. Progress is checkpointed.

11 Limitations and Future Work

1. **Simulation only.** The hardware executor exists but is disabled by default. Real-device runs require IBM Quantum credentials and the `qiskit-ibm-runtime` package.
2. **No distillation.** Rung 5 identifies factory-relevant heuristics, but the system does not build or evaluate distillation circuits. Extending to a 15-to-1 protocol is future work.
3. **No parallelism.** Experiments run sequentially. A process-level sharding of the challenger set is straightforward but not implemented.
4. **Small code.** The $[[4, 2, 2]]$ code has distance 2 and cannot correct errors. Scaling to the Steane code ($[[7, 1, 3]]$) or surface codes requires extending the circuit builder, but the search engine is code-agnostic.
5. **Witness is not fidelity.** The witness W is a necessary condition for magic-state quality, not sufficient. A full tomographic characterisation would be more rigorous but is orders of magnitude more expensive.
6. **No LLM in the loop.** The “auto” is algorithmic (statistics + guided search), not generative. Integrating an LLM to propose novel circuit architectures—rather than just varying parameters of a fixed family—is a natural extension.

12 Conclusion

AUTORESEARCH-QUANTUM demonstrates that the Karpathy autoresearch pattern transfers naturally to quantum computing experiments. The key ingredients are:

- A well-defined scalar objective that balances quality and cost.
- A combinatorial search space of physically motivated choices.
- A multi-strategy challenger generator that balances exploitation (NeighborWalk), exploration (RandomCombo), and learning (LessonGuided).
- A feedback loop that converts experiment results into machine-readable rules, narrows the search space, and propagates knowledge across rungs.
- A monotone ratchet that guarantees the incumbent never regresses.

The system’s output is not just a circuit—it is a structured body of experimental evidence, human-readable lessons, and machine-readable directives that a researcher can build on without re-running the search.

All code, configurations, and tests are available at <https://github.com/saymrwulf/autoresearch-quantum>. The system requires Python ≥ 3.11 , Qiskit ≥ 2.3 , and Qiskit Aer ≥ 0.17 .

References

- [1] S. Bravyi and A. Kitaev, “Universal quantum computation with ideal Clifford gates and noisy ancillas,” *Phys. Rev. A*, vol. 71, no. 2, p. 022316, 2005.
- [2] A. Karpathy, “Autoresearch,” <https://github.com/karpathy/autoresearch>, 2025.
- [3] Qiskit contributors, “Qiskit: An open-source framework for quantum computing,” <https://github.com/Qiskit/qiskit>, 2024.
- [4] D. Gottesman, “Stabilizer codes and quantum error correction,” PhD thesis, California Institute of Technology, 1997.
- [5] E. Knill, “Quantum computing with realistically noisy devices,” *Nature*, vol. 434, pp. 39–44, 2005.
- [6] E. T. Campbell, B. M. Terhal, and C. Vuillot, “Roads towards fault-tolerant universal quantum computation,” *Nature*, vol. 549, pp. 172–179, 2017.
- [7] D. Litinski, “Magic state distillation: Not as costly as you think,” *Quantum*, vol. 3, p. 205, 2019.