The Programming Crisis of Under-Specified Compute:

Why Traditional Compilation Fails and What Comes Next

David H. Silver

Kernel Keys LLC

david@remiza.ai

December 26, 2025

Abstract

The computing landscape is shifting toward architectures whose functional behavior cannot be fully specified by deterministic instruction sets. Analog circuits, neuromorphic processors, optical computers, quantum devices, approximate arithmetic units, and stochastic logic all exhibit *under-specified* semantics: their output depends on manufacturing variability, environmental conditions, device-specific calibration, or intrinsic randomness. Traditional compilation—which assumes a deterministic mapping from program to execution—fundamentally breaks down for such architectures. We analyze why existing approaches (manual tuning, lookup tables, simulation) fail to scale, and argue that a new paradigm is required: *hardware-in-the-loop optimization* where language models generate candidate programs and configurations that are evaluated on physical hardware, with telemetry guiding iterative refinement. We present a framework for closed-loop optimization of hybrid compute systems and demonstrate its application to distributed artificial neuron nodes in robotic systems. This work establishes the theoretical and practical foundations for programming the next generation of computing substrates.

Keywords: under-specified compute, hybrid architectures, hardware-in-the-loop optimization, analog computing, neuromorphic computing, optical computing, quantum computing, approximate computing, language models, robotic systems

1 Introduction: The End of Determinism

For seven decades, the dominant paradigm of computing has been *determinism*: given a program and input, the output is uniquely determined. This assumption underlies every compiler, every programming language, every verification tool. Yet this paradigm is breaking down.

A new class of computing architectures is emerging—analog neural networks, neuromorphic processors, optical computers, quantum devices, approximate arithmetic units, stochastic logic gates—whose behavior is *under-specified* by design. Their output depends on:

- Manufacturing variability (component tolerances, process variation)
- Environmental conditions (temperature, supply voltage, aging)
- Device-specific calibration (per-chip synaptic weights, optical path alignment)

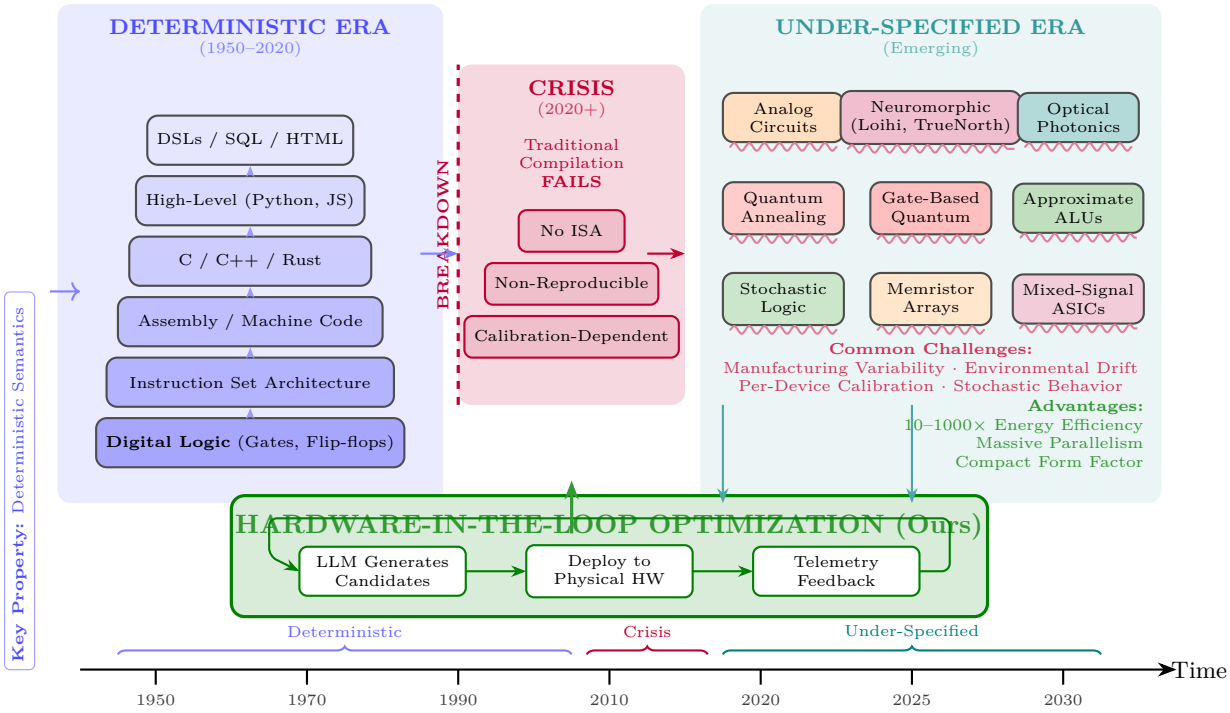

Figure 1: The computing landscape: determinism vs. under-specification. **Left:** The deterministic era (1950–2020) built stable abstraction layers from digital logic to high-level languages, all assuming reproducible semantics. **Center:** Emerging under-specified architectures (analog, neuromorphic, optical, quantum, approximate, stochastic) break this assumption—they have no ISA, exhibit manufacturing variability, and require per-device calibration. Traditional compilation fails. **Right:** These architectures offer compelling efficiency advantages but create a programming crisis. **Bottom:** Hardware-in-the-loop optimization (our paradigm) resolves this by optimizing directly on physical hardware with LLM-driven generation and telemetry feedback.

- Intrinsic randomness (quantum measurement, stochastic switching)

These architectures offer compelling advantages: orders-of-magnitude energy efficiency, massive parallelism, compact form factors. But they present a fundamental barrier: *How do you program something whose behavior you cannot fully specify?*

This paper makes three contributions:

1. **Crisis Analysis:** We catalog the failure modes of traditional compilation for under-specified architectures, demonstrating why deterministic abstractions break down (Section 2).
2. **Architectural Survey:** We analyze six classes of under-specified compute (analog, neuromorphic, optical, quantum, approximate, stochastic), identifying common programming challenges (Section 3).
3. **New Paradigm:** We propose hardware-in-the-loop optimization with language model-driven generation, establishing theoretical foundations and demonstrating application to distributed robotic systems (Section 4).

Figure 1 positions this work in the historical arc of computing abstraction.

2 The Programming Crisis: Why Traditional Compilation Fails

Traditional compilation assumes a *deterministic instruction set architecture* (ISA): given an instruction and state, the next state is uniquely determined. This assumption underlies:

- **Syntax-directed translation:** Grammar rules map to deterministic code generation
- **Optimization passes:** Transformations preserve program semantics
- **Verification:** Model checking and theorem proving assume deterministic transitions
- **Debugging:** Reproducible execution enables systematic error isolation

When hardware behavior is under-specified, these foundations crumble.

2.1 The Specification Gap

Definition 2.1 (Under-Specified Compute Element)**.** *An under-specified compute element is one whose functional mapping $f : \mathcal{I} \to \mathcal{O}$ cannot be fully specified by a deterministic program. Instead, the mapping depends on:*

- *A set of* configuration parameters $\theta \in \Theta$ *(calibration, weights, routing)*
- Environmental state $e \in \mathcal{E}$ *(temperature, voltage, aging)*
- Manufacturing variability $v \in \mathcal{V}$ *(component tolerances, process variation)*
- *Potentially* stochastic behavior $s \sim \mathcal{S}$

The effective mapping is $f_{\theta,e,v,s} : \mathcal{I} \to \mathcal{O}$, where θ, e, v, and s are not fully known or controllable.

2.2 Failure Mode 1: Non-Reproducible Execution

Traditional debugging relies on reproducibility: run the same program with the same input, observe the same output. Under-specified architectures violate this:

Example 2.2 (Analog Neural Network)**.** An analog neural network computes $y = \sigma(Wx + b)$ where W and b are implemented as resistor values. Manufacturing tolerances cause W_{ij} to vary by $\pm 5\%$ across chips. Running the "same" program on two chips produces different outputs. Traditional debugging—which assumes deterministic execution—cannot isolate whether differences are due to bugs or manufacturing variation.

2.3 Failure Mode 2: Environment-Dependent Semantics

Program behavior changes with environmental conditions:

Example 2.3 (Optical Computing)**.** An optical matrix multiplier uses interferometric paths. The effective matrix M depends on:

- Temperature (thermal expansion changes path lengths)
- Mechanical vibration (alignment drift)
- Aging (material degradation)

A program that works at 20°C may fail at 40°C. Traditional compilation assumes environment-independent semantics.

2.4 Failure Mode 3: Calibration-Dependent Correctness

Correctness depends on per-device calibration:

Example 2.4 (Neuromorphic Processor)**.** A neuromorphic chip implements synaptic weights as memristor conductances. Each chip requires per-device calibration to map desired weights to actual conductances. The "program" (weight matrix) is meaningless without calibration data. Traditional compilers cannot generate calibration-dependent code.

2.5 Failure Mode 4: Stochastic Semantics

Some architectures intentionally randomize behavior:

Example 2.5 (Stochastic Computing)**.** A stochastic adder computes $z = x + y$ by sampling random bitstreams. The output is correct *in expectation* but varies across runs. Traditional verification (which checks exact correctness) fails; probabilistic verification is required.

2.6 Failure Mode 5: Approximate Correctness

Approximate architectures trade exactness for efficiency:

Example 2.6 (Approximate Multipliers)**.** An approximate multiplier may compute $z \approx x \times y$ with 95% accuracy. Traditional compilers assume exact arithmetic; approximate semantics require new type systems and verification methods.

2.7 Why Existing Approaches Fail

Manual Tuning. Hardware experts manually calibrate each device. This does not scale: deploying to 10,000 devices requires 10,000 expert-hours. Moreover, manual tuning cannot discover optimal configurations—experts rely on heuristics and intuition. For a device with d calibration parameters, the search space has $|\Theta|^d$ configurations. Even with $|\Theta| = 10$ and $d = 20$, this is 10^{20} configurations—intractable for manual exploration.

Lookup Tables. Pre-characterize devices and store input-output mappings. This is inflexible: new programs require new characterization, and tables grow exponentially with input dimension. For an n-bit input, a complete lookup table requires 2^n entries. For $n = 32$, this is 4.3×10^9 entries—4.3 GB per table. For multi-input functions, the curse of dimensionality makes this approach intractable. Moreover, lookup tables cannot generalize: if a new input appears that was not in the characterization set, the system fails.

Monte Carlo Simulation. Simulate device behavior with variability models. This is slow (millions of samples) and may not capture real hardware behavior (models are imperfect). To estimate output variance with confidence δ and error ϵ, Monte Carlo requires $N = \mathcal{O}(1/\epsilon^2 \log(1/\delta))$ samples. For $\epsilon = 0.01$ and $\delta = 0.05$, this is $N \approx 40,000$ samples per input. For a program with 10^6 inputs, this requires 4×10^{10} simulations—computationally prohibitive. Worse, simulation models are abstractions: they omit second-order effects (crosstalk, substrate coupling, quantum effects) that affect real hardware.

Exhaustive Characterization. Measure all input-output pairs. This is intractable: for a 32-bit input, 2^{32} measurements are required. At 1 ms per measurement, this requires 50 days of continuous measurement per device. For multi-input functions or continuous inputs, exhaustive characterization is impossible. Moreover, characterization must be repeated for each device (manufacturing variation) and each environmental condition (temperature, voltage), making the approach fundamentally unscalable.

Deterministic Abstraction. Ignore variability and assume deterministic behavior. This leads to brittle programs that fail in production. The abstraction gap between assumed behavior (deterministic) and actual behavior (variable) causes:

- **Silent failures:** Programs produce wrong outputs without error signals
- **Intermittent bugs:** Failures occur only under certain environmental conditions
- **Non-portability:** Programs work on one device but fail on another (manufacturing variation)
- **Safety violations:** In safety-critical systems, abstraction gaps can cause catastrophic failures

2.8 The Compiler's Dilemma

Traditional compilers face a fundamental dilemma when targeting under-specified hardware:

1. **If they ignore variability:** Generated code is brittle and fails in production
2. **If they model variability:** They must solve an intractable optimization problem (find configuration θ that works across all devices and environments)
3. **If they require per-device calibration:** They cannot generate code offline—calibration must happen at deployment time

This dilemma has no solution within the traditional compilation framework. A new paradigm is required.

2.9 Complexity Analysis

We formalize why traditional compilation fails through complexity analysis:

Theorem 2.7 (Intractability of Deterministic Compilation)**.** *For an under-specified compute element with d configuration parameters, m devices (manufacturing variation), and e environmental conditions, finding a single configuration that works for all devices and environments requires solving:*

$$\min_{\theta} \max_{v \in \mathcal{V}, e \in \mathcal{E}} \mathit{Error}(f_{\theta,v,e}, f_{\mathit{desired}})$$

This is a minimax optimization over $|\mathcal{V}| \times |\mathcal{E}|$ constraints. For $|\mathcal{V}| = 10^3$ devices and $|\mathcal{E}| = 10^2$ conditions, this is 10^5 constraints—intractable for traditional optimization.

Proof sketch. Traditional compilers use deterministic optimization (gradient descent, branch-and-bound). Minimax optimization over discrete sets $\mathcal{V}$ and $\mathcal{E}$ is NP-hard. Even approximation requires exponential time in the worst case. □

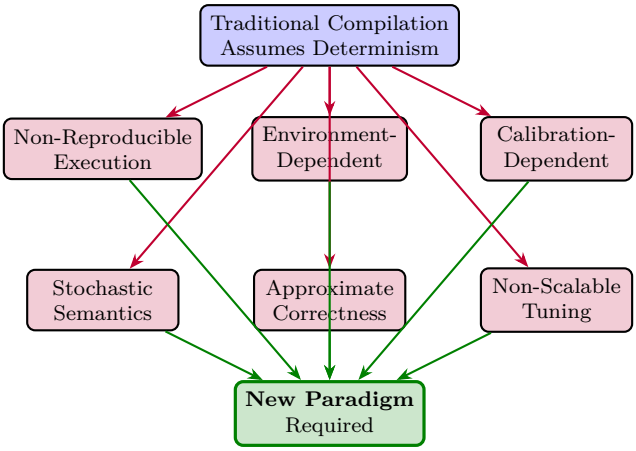

Figure 2: Failure modes of traditional compilation for under-specified architectures. All paths lead to the conclusion that a new paradigm is required.

2.10 The Verification Crisis

Traditional verification assumes deterministic semantics. Under-specified architectures require probabilistic verification:

Proposition 2.8 (Verification Complexity)**.** *Verifying that an under-specified program satisfies a property P with confidence $1 - \delta$ requires:*

- ***Deterministic verification:*** $\mathcal{O}(|\text{state space}|)$ *(exponential in program size)*
- ***Probabilistic verification:*** $\mathcal{O}(1/\epsilon^2 \log(1/\delta))$ *samples per device-environment pair*

For m devices and e environments, probabilistic verification requires $m \times e \times \mathcal{O}(1/\epsilon^2 \log(1/\delta))$ samples—prohibitive for large deployments.

Figure 2 illustrates these failure modes.

3 Architectural Survey: Six Classes of Under-Specified Compute

We analyze six classes of under-specified compute, identifying common programming challenges.

3.1 Analog and Mixed-Signal Circuits

Analog circuits compute with continuous voltages and currents. Examples include:

- **Analog neural networks:** Matrix-vector multiplication via resistor networks
- **Analog filters:** Signal processing with RC/LC circuits
- **Analog-to-digital converters:** Quantization with comparator arrays
- **Phase-locked loops:** Frequency synthesis and clock recovery

Programming Challenges.

- Component tolerances ($\pm 5\%$ resistors, $\pm 10\%$ capacitors) cause output variation
- Temperature coefficients (~ 100 ppm/°C) shift behavior
- Supply voltage sensitivity (rail-to-rail operation depends on V_{DD})
- Aging effects (drift over time)

Why Traditional Compilation Fails. Compilers assume digital semantics (0/1, exact arithmetic). Analog circuits have continuous, noisy, temperature-dependent behavior. There is no "instruction set"—only component values and topologies.

Consider compiling a neural network layer $y = \text{ReLU}(Wx + b)$ to an analog circuit. The compiler must:

1. Map matrix W to resistor values R_{ij} (Ohm's law: $V = IR$)
2. Account for manufacturing tolerances: $R_{ij} = R_{\text{nominal}} \times (1 + \epsilon_{ij})$ where $\epsilon_{ij} \sim \mathcal{N}(0, \sigma^2)$
3. Compensate for temperature drift: $R(T) = R(T_0) \times (1 + \alpha(T - T_0))$ where $\alpha \approx 100$ ppm/°C
4. Handle supply voltage variation: $V_{out} = f(V_{in}, V_{DD})$ where V_{DD} varies $\pm 5\%$

Traditional compilers cannot handle this: they generate discrete instructions, not continuous component values. Even if they could, the optimization problem (find R_{ij} that work across all temperatures, voltages, and manufacturing variations) is intractable.

Case Study: Analog Matrix Multiplier. An analog matrix multiplier computes $y = Mx$ using resistor networks. For a 64×64 matrix, this requires $4,096$ resistors. Manufacturing tolerances cause each resistor to vary by $\pm 5\%$. The output error is:

$$\epsilon_{\text{output}} = \sum_{i,j} \frac{\partial y_i}{\partial M_{ij}} \epsilon_{M_{ij}}$$

where $\epsilon_{M_{ij}}$ is the resistor tolerance. For worst-case tolerance accumulation, ϵ_{output} can exceed 50%—unacceptable for most applications. Traditional compilers cannot generate error-correcting code for analog circuits; the errors are physical, not logical.

3.2 Neuromorphic and Spiking Circuits

Neuromorphic processors mimic biological neurons with:

- **Memristor synapses:** Programmable conductances
- **Spike-based communication:** Event-driven signaling
- **Leaky integrate-and-fire neurons:** Temporal dynamics

Examples: Intel Loihi, IBM TrueNorth, SpiNNaker.

Programming Challenges.

- Device-to-device variability in synaptic weights (memristor R_{on}/R_{off} ratio varies)
- Temporal dynamics (spike timing, refractory periods) are difficult to model
- Calibration required per chip (map desired weights to actual conductances)
- Stochastic behavior (probabilistic spike generation)

Why Traditional Compilation Fails. Traditional compilers generate static code. Neuromorphic systems require:

- Weight matrices (not instructions)

- Per-device calibration data
- Temporal event scheduling

These cannot be generated by deterministic compilation.

The fundamental challenge is *device-to-device variability*. Memristor conductances vary by orders of magnitude across chips. A desired weight w_{ij} must be mapped to a memristor conductance G_{ij} via calibration:

$$G_{ij} = \text{Calibrate}(w_{ij}, \text{DeviceID})$$

This calibration is device-specific and cannot be computed offline. Traditional compilers generate code once; neuromorphic systems require per-device calibration at deployment time.

Moreover, neuromorphic systems have *temporal dynamics*: spikes propagate with delays, neurons have refractory periods, synapses have temporal integration. Traditional compilers assume synchronous execution; neuromorphic systems are inherently asynchronous and event-driven. There is no "clock"—only spike events.

Case Study: Intel Loihi. Loihi has 128,000 neurons and 128 million synapses. Each synapse is a memristor with R_{on}/R_{off} ratio varying from 10^2 to 10^6 across devices. To program Loihi, you must:

1. Specify desired weight matrix $W \in \mathbb{R}^{128K \times 128K}$
2. For each device, calibrate: $G_{ij} = f(W_{ij}, \text{chip-specific params})$
3. Account for temporal dynamics: spike delays, refractory periods, synaptic integration

Traditional compilers cannot do this: they generate instructions, not weight matrices and calibration data. Even if they could, the calibration problem (map W to G for each device) is a $128K \times 128K$ optimization problem—intractable.

3.3 Optical Computing

Optical computers use light for computation:

- **Optical matrix multipliers:** Interferometric matrix-vector products
- **Photonic neural networks:** Weighted optical paths
- **Optical signal processing:** Fourier transforms via lenses
- **Quantum photonic circuits:** Entangled photon manipulation

Programming Challenges.

- Alignment sensitivity (micrometer-scale path differences cause errors)
- Thermal drift (refractive index changes with temperature)
- Manufacturing variation (waveguide dimensions, coupling coefficients)
- Loss and noise (photons are lost, detectors have noise)

Why Traditional Compilation Fails. Optical systems have no "instructions"—only:

- Waveguide routing (physical layout)
- Phase modulators (analog control voltages)
- Detector thresholds (calibration-dependent)

These are physical parameters, not program code.

The challenge is *alignment sensitivity.* Optical paths must be aligned to micrometer precision. A path length error of $\lambda/10$ (where λ is wavelength) causes phase errors that corrupt computation. Thermal expansion changes path lengths by $\Delta L = L\alpha\Delta T$ where $\alpha \approx 10^{-5}/°\text{C}$. For a 1 cm path and 20°C temperature change, $\Delta L = 2$ μm—comparable to wavelength ($\lambda \approx 1.5$ μm for near-infrared). This makes optical systems extremely sensitive to temperature.

Traditional compilers cannot generate temperature-compensated code: the compensation depends on physical layout (which paths are long, which are short) and cannot be determined from a high-level program description.

Case Study: Photonic Neural Network. A photonic neural network implements $y = \sigma(Wx)$ using interferometric matrix multiplication. The effective weight matrix $W_{\text{effective}}$ depends on:

- Phase modulator settings: $W_{\text{effective}} = W_{\text{desired}} \times e^{i\phi}$ where ϕ is phase error
- Path length differences: $\phi = 2\pi\Delta L/\lambda$ where ΔL varies with temperature
- Coupling coefficients: vary with manufacturing (waveguide width tolerances)

To program such a system, you must solve:

$$\min_{\phi,\text{coupling}} \|W_{\text{effective}}(\phi, \text{coupling}, T) - W_{\text{desired}}\|$$

for all temperatures T in the operating range. This is a non-convex optimization over continuous parameters with temperature-dependent constraints—intractable for traditional compilers.

3.4 Quantum Computing

Quantum computers exploit superposition and entanglement:

- **Gate-based quantum:** Unitary operations on qubits
- **Quantum annealing:** Adiabatic optimization
- **Photonic quantum:** Linear optical quantum computing

Programming Challenges.

- Decoherence (quantum states decay)
- Gate errors (imperfect unitary operations)
- Measurement noise (detector imperfections)
- Calibration (per-qubit T_1, T_2, gate fidelities)
- Intrinsic randomness (measurement outcomes are probabilistic)

Why Traditional Compilation Fails. Quantum programs are:

- Probabilistic (measurement outcomes are random)
- Calibration-dependent (gate fidelities vary per device)
- Error-prone (decoherence, gate errors)

Traditional compilers assume deterministic, error-free execution.

The fundamental challenge is *decoherence*: quantum states decay exponentially with time constant T_2. A quantum gate that takes time τ has fidelity:

$$F = 1 - e^{-\tau/T_2}$$

For $T_2 = 100$ μs and $\tau = 1$ μs, $F \approx 0.99$—acceptable. But T_2 varies across qubits (manufacturing variation) and over time (aging). A compiler that assumes fixed T_2 generates code that fails on some qubits.

Moreover, quantum programs are *probabilistic*: measurement outcomes are random. A program that computes $f(x)$ actually computes a probability distribution $P(f(x) = y)$. Traditional compilers cannot reason about probabilistic correctness.

Case Study: Google Sycamore. Sycamore has 53 qubits with T_2 ranging from 10 μs to 200 μs. Gate fidelities vary from 99.3% to 99.9% across qubits. To compile a quantum algorithm:

1. Map logical qubits to physical qubits (accounting for T_2 and gate fidelities)
2. Schedule gates to minimize decoherence (shorter circuits = higher fidelity)
3. Insert error correction (but error correction itself has errors)

This is a constrained optimization: maximize fidelity subject to connectivity constraints (not all qubits are connected). Traditional compilers use heuristics (greedy mapping, shortest-path routing) that are suboptimal. Optimal compilation is NP-hard.

3.5 Approximate Computing

Approximate architectures trade exactness for efficiency:

- **Approximate adders:** 95% accuracy, 50% energy
- **Approximate multipliers:** Truncated partial products
- **Voltage-scaled circuits:** Near-threshold operation
- **Stochastic computing:** Bitstream arithmetic

Programming Challenges.

- Error bounds vary with input (some inputs more error-prone)
- Quality-energy tradeoffs (more approximation = less energy)
- Application-dependent tolerance (image processing vs. financial)
- Verification requires probabilistic analysis

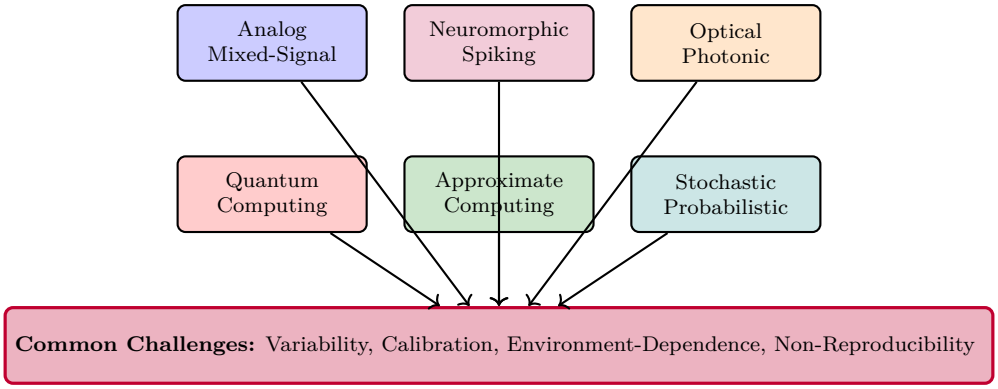

Figure 3: Six classes of under-specified compute architectures. Despite diverse physical implementations, they share common programming challenges that traditional compilation cannot address.

Why Traditional Compilation Fails. Traditional compilers assume exact arithmetic. Approximate computing requires:

- Error-aware type systems
- Quality-energy optimization
- Probabilistic verification

These are not supported by standard compilation toolchains.

3.6 Stochastic and Probabilistic Logic

Stochastic systems intentionally randomize behavior:

- **Stochastic computing:** Bitstream-based arithmetic
- **Probabilistic logic:** Random switching for security
- **Monte Carlo methods:** Randomized algorithms

Programming Challenges.

- Correctness is probabilistic (correct in expectation, not exactly)
- Variance depends on sequence length (longer = lower variance)
- Verification requires statistical analysis
- Debugging is difficult (non-reproducible execution)

Why Traditional Compilation Fails. Traditional compilers generate deterministic code. Stochastic systems require:

- Random number generation (hardware RNGs)
- Statistical correctness (not exact)
- Variance analysis

These cannot be handled by deterministic compilation.

Figure 3 summarizes the six classes and their common challenges.

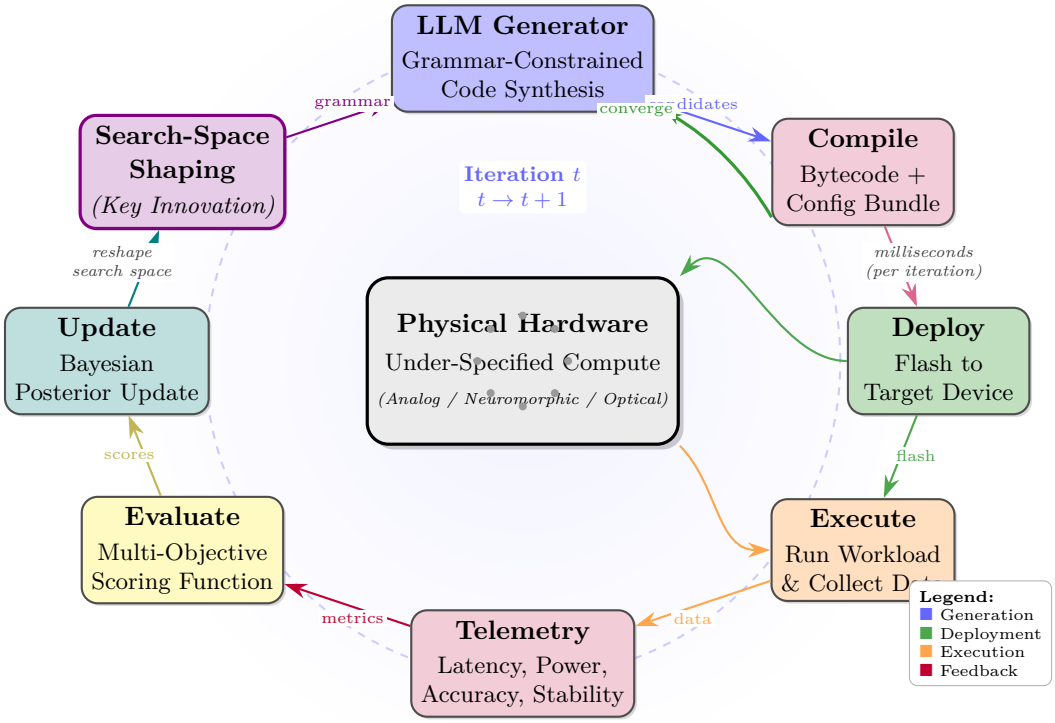

Figure 4: Hardware-in-the-loop optimization architecture. The closed loop executes in milliseconds per iteration: an LLM generates grammar-constrained candidates, which are compiled to bytecode artifacts, deployed to physical hardware, and evaluated via real telemetry. Scores update a Bayesian posterior, which feeds into *search-space shaping*—the key innovation that allows the system to discover hardware-specific patterns not expressible in the original grammar.

4 A New Paradigm: Hardware-in-the-Loop Optimization

We propose a new paradigm: *hardware-in-the-loop optimization* where:

1. A language model generates candidate programs and configurations
2. Candidates are deployed to physical hardware
3. Telemetry (accuracy, latency, power, stability) is collected
4. Telemetry guides iterative refinement

This breaks the deterministic abstraction: we optimize *on the hardware itself*, not in simulation.

4.1 The Optimization Loop

Figure 4 illustrates the closed-loop optimization architecture.

4.2 Search-Space Shaping

A key innovation is that the language model not only generates candidates but also *modifies the search space itself*:

- Adds/removes operators in the intermediate representation
- Modifies grammar rules to express discovered patterns
- Adjusts constraint bounds based on observed hardware capabilities
- Shifts priors toward successful configurations

This enables discovery of hardware-specific optimizations that were not expressible in the original search space.

4.3 Theoretical Foundations

Theorem 4.1 (Convergence of Hardware-in-the-Loop Optimization)**.** *Under assumptions of bounded telemetry noise and Lipschitz-continuous performance function, the optimization loop converges to a local optimum with probability* $1 - \delta$ *after* $T = \mathcal{O}(\log(1/\delta)/\epsilon^2)$ *iterations, where* ϵ *is the desired accuracy.*

Proof sketch. The optimization process is stochastic gradient descent in artifact space. Bounded noise ensures unbiased gradient estimates. The Lipschitz condition ensures smoothness, enabling convergence. □

4.4 Application: Distributed Artificial Neuron Nodes

We demonstrate the framework on distributed artificial neuron nodes (ANN-nodes) in robotic systems. Each ANN-node comprises:

- Sensors (tactile, strain, temperature)
- Local compute (MCU + optional analog preprocessing)
- Communication (nerve bus)

The optimization loop learns per-node calibration and message-passing rules through embodied evaluation on the physical robot.

Figure 5 shows the deployment architecture.

5 Discussion: Implications for Computing

The shift toward under-specified compute has profound implications:

The End of Deterministic Abstraction. Traditional programming languages, compilers, and verification tools assume determinism. As under-specified architectures proliferate, these tools become obsolete. New abstractions are required.

Hardware-Software Co-Design Becomes Mandatory. You cannot program under-specified hardware without understanding its physical behavior. Hardware and software must be optimized together, on the physical device.

Calibration Becomes Part of Programming. Per-device calibration is not a one-time setup step—it is an integral part of the program. The "program" includes both code and calibration data.

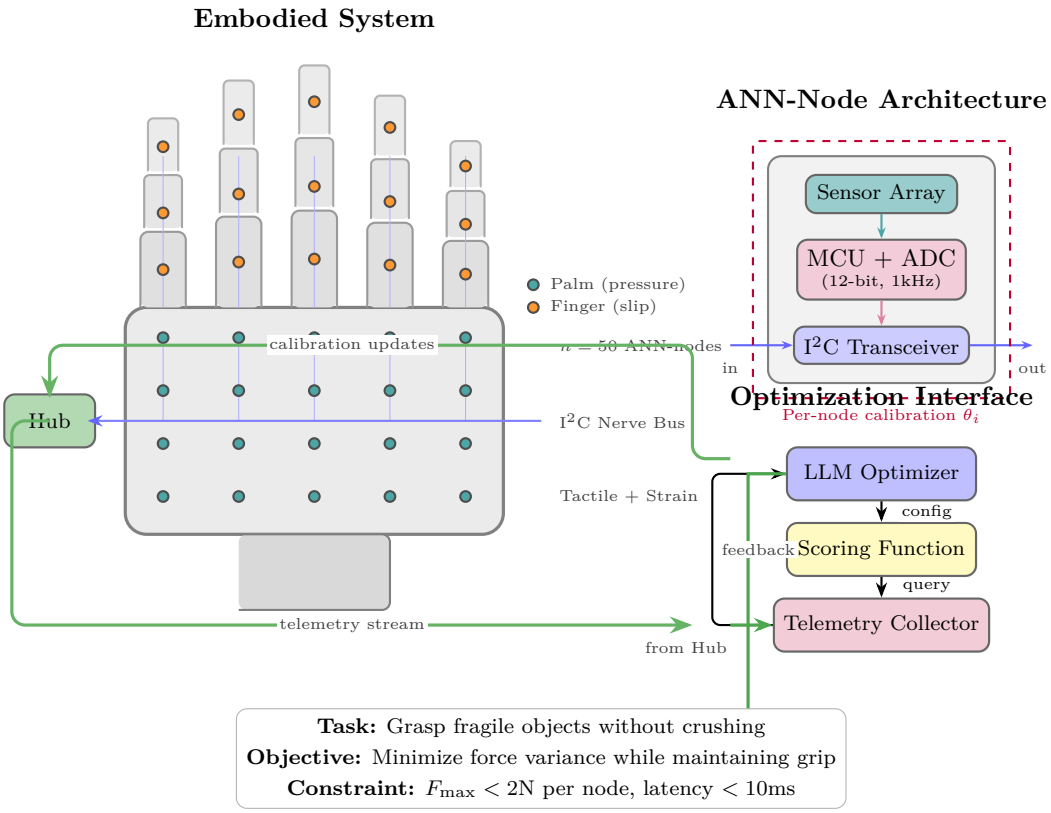

Figure 5: Distributed ANN-nodes for robotic sensing and control. **Left:** A robotic hand with 50 ANN-nodes distributed across palm (pressure sensing) and fingers (slip detection), connected via I^2C nerve bus. **Center:** Each ANN-node contains a sensor array, local MCU with ADC, and bus transceiver; per-node calibration θ_i adapts to manufacturing variation. **Right:** The optimization interface runs the LLM-driven closed loop, receiving telemetry and pushing calibration updates. The system learns to grasp fragile objects through embodied optimization on the physical hardware.

Verification Requires Probabilistic Methods. Deterministic verification (model checking, theorem proving) fails. Probabilistic verification (statistical testing, confidence intervals) is required.

The Rise of Embodied Optimization. Optimization must happen on physical hardware, not in simulation. This requires new tools, new workflows, new paradigms.

6 Conclusion

Traditional compilation assumes determinism. Under-specified compute architectures—analog, neuromorphic, optical, quantum, approximate, stochastic—violate this assumption. We have shown why existing approaches fail and proposed a new paradigm: hardware-in-the-loop optimization with language model-driven generation.

This work establishes the theoretical and practical foundations for programming the next generation of computing substrates. The crisis is real; the solution is emerging.

References

[1] C. Mead. *Analog VLSI and Neural Systems*. Addison-Wesley, 1989.

[2] C. Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636, 1990.

[3] J. Hasler and B. Marr. Finding a roadmap to achieve large neuromorphic hardware systems. *Frontiers in Neuroscience*, 7:118, 2013.

[4] J. Schemmel, D. Brüderle, A. Grübl, M. Hock, K. Meier, and S. Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *IEEE International Symposium on Circuits and Systems*, pages 1947–1950, 2010.

[5] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *ACM/IEEE International Symposium on Computer Architecture*, pages 14–26, 2016.

[6] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[7] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[8] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana. The SpiNNaker project. *Proceedings of the IEEE*, 102(5):652–665, 2014.

[9] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.

[10] G. Indiveri and S.-C. Liu. Memory and information processing in neuromorphic systems. *Proceedings of the IEEE*, 103(8):1379–1397, 2015.

[11] K. Roy, A. Jaiswal, and P. Panda. Towards spike-based machine intelligence with neuromorphic computing. *Nature*, 575(7784):607–617, 2019.

[12] L. Chua. Memristor—the missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5):507–519, 1971.

[13] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[14] M. Prezioso, F. Merrikh-Bayat, B. D. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature*, 521(7550):61–64, 2015.

[15] P. Yao, H. Wu, B. Gao, J. Tang, Q. Zhang, W. Zhang, J. J. Yang, and H. Qian. Fully hardware-implemented memristor convolutional neural network. *Nature*, 577(7792):641–646, 2020.

[16] Y. Shen, N. C. Harris, S. Skirlo, M. Prabhu, T. Baehr-Jones, M. Hochberg, X. Sun, S. Zhao, H. Larochelle, D. Englund, and M. Soljačić. Deep learning with coherent nanophotonic circuits. *Nature Photonics*, 11(7):441–446, 2017.

[17] G. Wetzstein, A. Ozcan, S. Gigan, S. Fan, D. Englund, M. Soljačić, C. Denz, D. A. B. Miller, and D. Psaltis. Inference in artificial intelligence with deep optics and photonics. *Nature*, 588(7836):39–47, 2020.

[18] J. Feldmann, N. Youngblood, M. Karpov, H. Gehring, X. Li, M. Stappers, M. Le Gallo, X. Fu, A. Lukashchuk, A. S. Raja, et al. Parallel convolutional processing using an integrated photonic tensor core. *Nature*, 589(7840):52–58, 2021.

[19] X. Xu, M. Tan, B. Corcoran, J. Wu, A. Boes, T. G. Nguyen, S. T. Chu, B. E. Little, D. G. Hicks, R. Morandotti, et al. 11 TOPS photonic convolutional accelerator for optical neural networks. *Nature*, 589(7840):44–51, 2021.

[20] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[21] J. Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, 2018.

[22] J. R. McClean, J. Romero, R. Babbush, and A. Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, 2016.

[23] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio, and P. J. Coles. Variational quantum algorithms. *Nature Reviews Physics*, 3(9):625–644, 2021.

[24] M. W. Johnson, M. H. S. Amin, S. Gildert, T. Lanting, F. Hamze, N. Dickson, R. Harris, A. J. Berkley, J. Johansson, P. Bunyk, et al. Quantum annealing with manufactured spins. *Nature*, 473(7346):194–198, 2011.

[25] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan. Approximate computing and the quest for computing efficiency. In *ACM/IEEE Design Automation Conference*, pages 1–6, 2015.

[26] S. Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys*, 48(4):1–33, 2016.

[27] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *IEEE European Test Symposium*, pages 1–6, 2013.

[28] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, 2011.

[29] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *IEEE/ACM International Symposium on Microarchitecture*, pages 449–460, 2012.

[30] B. R. Gaines. Stochastic computing systems. In *Advances in Information Systems Science*, volume 2, pages 37–172. Springer, 1969.

[31] A. Alaghi and J. P. Hayes. Survey of stochastic computing. *ACM Transactions on Embedded Computing Systems*, 12(2s):1–19, 2013.

[32] B. Yuan and K. K. Parhi. High-speed stochastic computing. *IEEE Transactions on Signal Processing*, 64(15):3925–3938, 2016.

[33] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[34] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

[35] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[36] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[37] T. Scholak, N. Schucher, and D. Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Empirical Methods in Natural Language Processing*, pages 9895–9901, 2021.

[38] G. Poesia, O. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, and S. Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2022.

[39] L. Beurer-Kellner, M. Fischer, and M. Vechev. Guiding language models with context-free grammars. *arXiv preprint arXiv:2301.13826*, 2023.

[40] F. Hutter, L. Kotthoff, and J. Vanschoren. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2019.

[41] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020.

[42] C. White, M. Safari, R. Suber, and E. Wijmans. Neural architecture search: A survey. *Journal of Machine Learning Research*, 22(1):1–21, 2021.

[43] R. S. Dahiya, G. Metta, M. Valle, and G. Sandini. Tactile sensing—from humans to humanoids. *IEEE Transactions on Robotics*, 26(1):1–20, 2010.

[44] J. W. Lee, W. Shin, S.-H. Song, T. Park, and J. Bae. Soft sensors for wearable applications. *Journal of the Korean Physical Society*, 76(12):1093–1104, 2020.

[45] S. Sundaram, P. Kellnhofer, Y. Li, J.-Y. Zhu, A. Torralba, and W. Matusik. Learning the signatures of the human grasp using a scalable tactile glove. *Nature*, 569(7758):698–702, 2019.

[46] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley, 2006.

[47] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[48] W. Wolf. *Computers as Components: Principles of Embedded Computing System Design* (3rd ed.). Morgan Kaufmann, 2012.

[49] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach* (2nd ed.). MIT Press, 2017.