Agent-based Deployment on FPGA-based Edge Devices

Mohammed Mshragi
MshragiM@cardiff.ac.uk
Cardiff University
School of Engineering
Cardiff, United Kingdom

Ioan Petri
PetriI@cardiff.ac.uk
Cardiff University
School of Engineering
Cardiff, United Kingdom

Othmane Belarbi
BelarbiO@cardiff.ac.uk
Cardiff University
School of Computational &
Mathematical Sciences
Cardiff, United Kingdom

Charith Perera
PereraC@cardiff.ac.uk
Cardiff University
School of Computational &
Mathematical Sciences
Cardiff, United Kingdom

Omer Rana
RanaOF@cardiff.ac.uk
Cardiff University
School of Computational &
Mathematical Sciences
Cardiff, United Kingdom

Abstract

Deploying Machine Learning (ML) models on edge devices typically requires manually selecting hardware, profiling resources, choosing toolchains, and subsequently undertaking tuning of the developed implementations—a process demanding expertise across both software and hardware domains. This paper presents an agent-based methodology that autonomously translates high-level user objectives into deployable edge-ML implementations from natural-language requests specifying goals such as latency, throughput, or accuracy. The agent discovers connected Field-Programmable Gate Array (FPGA)-based devices, profiles their resources, selects an implementation strategy, generates FPGA hardware artefacts, deploys them to physical targets, and validates performance using measured on-device metrics. We evaluate the methodology on a ZCU104 SoC FPGA and a PYNQ-Z1 FPGA using an HVAC energy-prediction task. The agent explores multiple quantisation levels and two FPGA implementation paths, achieving up to 3.83 million inferences per second on the ZCU104, corresponding to a 37.5% throughput improvement over the automated hls4ml baseline.

CCS Concepts

• **Hardware → Reconfigurable logic and FPGAs**; • **Computer systems organization → Embedded systems**; • **Computing methodologies** → *Intelligent agents.*

Keywords

Agentic AI, Hardware Synthesis, Dynamic Deployment

ACM Reference Format:

Mohammed Mshragi, Ioan Petri, Othmane Belarbi, Charith Perera, and Omer Rana. 2018. Agent-based Deployment on FPGA-based Edge Devices. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Deploying Machine Learning (ML) workloads at the edge can reduce communication latency, limit the transfer of sensitive data, and support applications that require local or real-time decision-making [4, 8]. However, translating an ML workload into an implementation that runs efficiently on an edge device remains a complex engineering task. Edge devices commonly operate under constrained computational, memory, and power budgets, requiring models and runtimes to be carefully adapted to the target hardware [1, 3]. Furthermore, edge environments increasingly incorporate Field-Programmable Gate Array (FPGA) and System-on-Chip Field-Programmable Gate Array (SoC FPGA) platforms alongside conventional processors and specialised accelerators. These targets differ substantially in their memory hierarchies, programming models, communication interfaces, software ecosystems, and deployment toolchains, increasing the complexity and engineering effort of manual deployment [2, 3].

Conventional edge-ML deployment requires developers to perform several interdependent tasks manually. They must inspect available hardware, determine whether the workload is compatible with each candidate target, select an appropriate implementation framework, configure device-specific toolchains, generate and transfer deployment artefacts, and validate the implementation on the physical device. This end-to-end pipeline requires expertise spanning model optimisation, embedded software development, hardware synthesis, interface design, device configuration, and runtime evaluation. Moreover, these decisions are interdependent: changes to model architecture affect both accuracy and resource consumption, while the choice of hardware target and memory architecture affects latency, throughput, and energy consumption [1, 6].

Existing frameworks automate important parts of this process. Compiler systems such as TVM provide model and operator optimisation across multiple processor and accelerator backends [2]. Embedded runtimes such as TensorFlow Lite Micro support inference on resource-constrained devices [3], while FPGA frameworks

such as FINN and hls4ml automate the generation of specialised hardware accelerators [5, 6]. However, these systems commonly assume that the target device, toolchain, or implementation flow has already been selected. Consequently, developers remain responsible for coordinating the wider deployment lifecycle, including discovering accessible hardware, profiling device capabilities, selecting suitable implementation flows, invoking synthesis and deployment toolchains, diagnosing failures, comparing measured outcomes, and refining the design.

Agentic Artificial Intelligence (AI) provides an opportunity to coordinate these fragmented activities as a unified engineering process. Unlike a fixed automation script, a Large Language Model (LLM)-driven agent can interpret high-level objectives, plan a sequence of actions, invoke specialised tools, observe intermediate outcomes, and revise its plan when a candidate implementation is infeasible or fails to satisfy the requested goals. Applying this capability to FPGA-based edge deployment could allow users to specify objectives such as low latency, high throughput, acceptable accuracy, limited resource use, or reduced implementation cost without manually determining the corresponding hardware implementation strategy. Nevertheless, realising this vision requires more than generating source code or hardware descriptions: the agent must discover and identify physical devices, reason over device capabilities, manage FPGA synthesis and deployment toolchains, deploy generated artefacts, and use measured runtime results to guide subsequent decisions.

This paper presents an agentic engineering system for automating and iteratively refining the deployment of ML models and workflows across FPGA-based edge hardware. The system transforms a high-level user request into a validated on-device implementation through a closed-loop process comprising request interpretation, device discovery, resource and capability profiling, target selection, implementation generation, deployment, runtime validation, and replanning. Target-specific functionality is exposed through modular backends, separating the agent's orchestration logic from the tools and artefacts required by individual FPGA platforms and implementation flows. A candidate deployment is accepted only after it has been executed and evaluated on the physical target; when its measured performance does not satisfy the user-defined objective, the agent may modify the model or implementation, select an alternative backend, or reconsider the deployment target. The main contributions of this paper are as follows: (i) an agent-based methodology for autonomously translating high-level deployment objectives into deployable edge-ML implementations; (ii) an end-to-end implementation on physical FPGA platforms covering device discovery, hardware generation, deployment, quantisation-aware design exploration, runtime validation, and on-device training; and (iii) an experimental evaluation using an HVAC energy-prediction workload on PYNQ-Z1 and ZCU104 FPGA platforms.

2 System Components

The orchestration agent was implemented using OpenAI Codex Command-Line Interface (CLI) v0.140.0 with the `gpt-5.5` model and `high` reasoning effort. Codex was launched from the experiment workspace on a host running Ubuntu 22.04.5 LTS. No application-specific agent framework, custom function-calling interface, or Model Context Protocol (MCP) server was introduced; the default Codex instructions and action interfaces were used.

Codex exposed shell execution and workspace inspection and editing as its principal actions. Through the shell, the agent could invoke programs installed on the host, including Linux device-discovery utilities, Python and machine-learning libraries, hls4ml, Vivado/Vitis HLS, compilation and synthesis scripts, and device-access programs such as `ssh`, `scp`, `rsync`, and serial-console utilities. Command output, exit status, generated files, synthesis reports, and runtime measurements were returned to the active agent context and informed subsequent actions. These host programs were accessed through the shell rather than exposed as separate agent-specific tools.

Figure 1 illustrates the overall architecture of the proposed system. The agent first interprets a user request and extracts the workload type, available inputs, and optimisation objectives, such as latency, throughput, accuracy, energy, memory footprint, implementation cost, or a weighted combination of these. It then searches for connected edge devices and establishes an execution path: local access is attempted first through USB, JTAG, UART, or serial interfaces. If unavailable, the agent falls back to network-based access through SSH/TLS over Ethernet, Wi-Fi or another network interface. Supporting both access routes enables the same orchestration approach to be applied across platforms that expose different programming, debugging, and deployment interfaces.

After device discovery, the agent profiles each candidate target. For FPGA and SoC FPGA devices, this covers programmable-logic resources such as Lookup Tables (LUTs), flip-flops, Digital Signal Processor (DSP) blocks, memory (BRAM/URAM, DMA availability, AXI interconnect), supported clock frequencies, board support files, synthesis tools, and overlay support. The architecture can be extended with target-specific profiles for other classes of edge hardware; however, the experimental evaluation in this work focuses on FPGA and SoC FPGA platforms.

The decision and ranking stage sits at the core of the methodology. The agent compares the profiled devices against the user objective and the baseline model, determines whether the task involves inference, training, or a hybrid update, and selects a target-specific implementation strategy. For the FPGA platforms considered in this work, the decision accounts for available hardware resources, supported synthesis flows, arithmetic precision, achievable parallelism, interface requirements, and user-defined constraints such as latency, throughput, accuracy, and resource utilisation.

Once a target is selected, the agent generates an implementation using an appropriate target-specific backend. In the FPGA evaluation presented in this work, the available paths include hls4ml-based generation, custom-HLS using Vivado/Vitis HLS, RTL components, and generated host-control software. For FPGA targets, the agent also determines the required data-movement structure, using AXI4-Lite for control registers, AXI4-Stream for streaming datapaths, AXI memory-mapped interfaces for addressable buffers, and DMA where high-bandwidth transfer between processor memory and programmable logic is required. These interface choices affect latency, throughput, resource utilisation, and software complexity.

The generated FPGA artefacts are then deployed to the selected physical device. These may include bitstreams, overlays, hardware

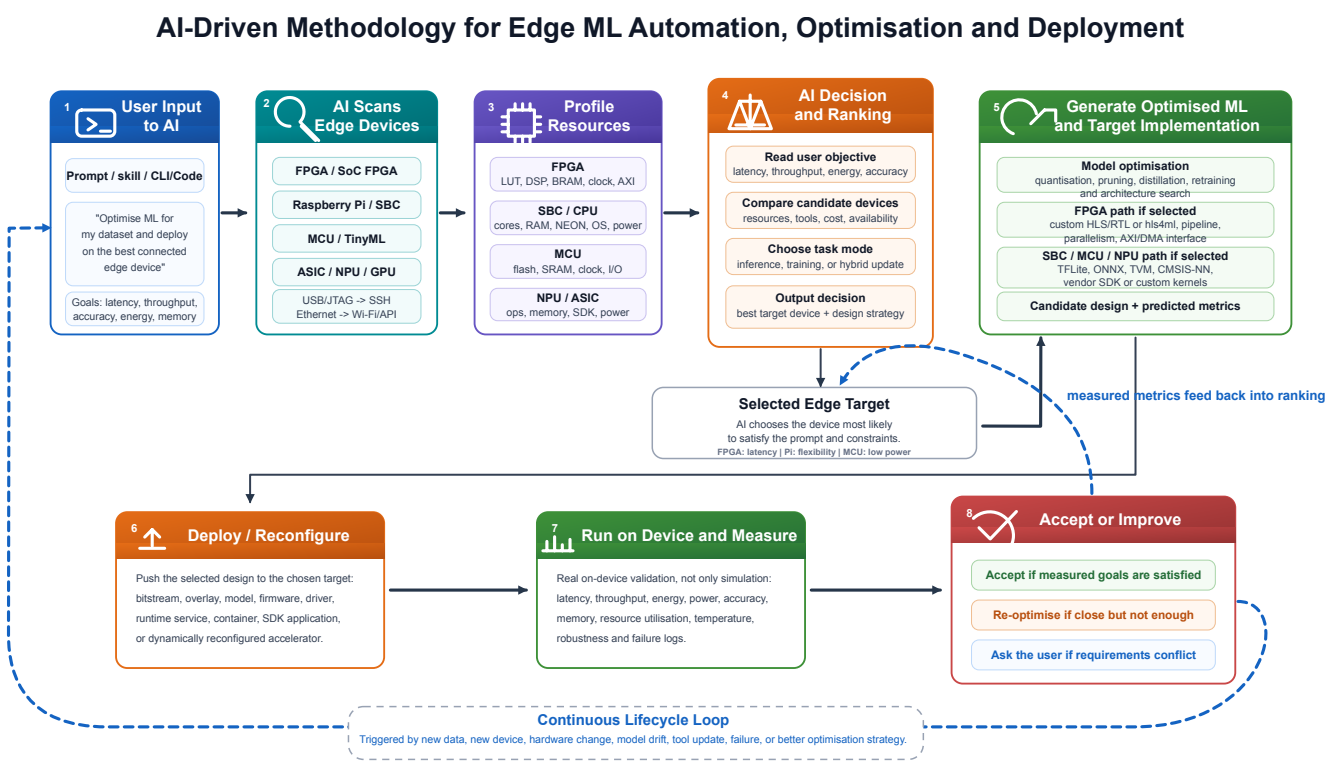


Figure 1: System-level methodology for agent-driven edge-ML deployment, including target discovery, implementation generation, physical deployment, runtime measurement, and iterative refinement.

metadata, drivers, and host-control scripts. The deployed implementation is executed on the target, and on-device metrics such as latency, throughput, accuracy, resource utilisation, and execution failures are returned to the agent. These measurements are compared with the user-defined objective or earlier candidate deployments. If the objective is not satisfied, the result feeds back into the optimisation loop, allowing the agent to revise the quantisation configuration, regenerate the hardware implementation, modify the interface structure, reconsider the target device, or request revised constraints.

3 Experimental Testbed

To evaluate the proposed system under representative deployment conditions, an edge-AI testbed was constructed using different deployment mechanisms and communication interfaces. The objective of the testbed was to evaluate the proposed agentic system's ability to autonomously discover available hardware, profile platform capabilities, select an appropriate deployment target, generate platform-specific implementations, and validate the resulting deployments on physical hardware.

Figure 2 illustrates the experimental platform. A host workstation acts as the orchestration node and executes the proposed AI agent. It receives high-level deployment requests, performs hardware discovery, invokes the required software and hardware toolchains, transfers generated artefacts to the selected target, and collects runtime measurements from deployed applications. The host therefore provides a unified execution environment while delegating inference or hardware acceleration tasks to the selected FPGA platform.

The experimental platform consists of two FPGA deployment targets. The ZCU104 represents a high-performance SoC FPGA platform capable of hardware acceleration through programmable logic. It supports hardware/software co-design workflows, high-level synthesis, custom RTL implementations, and heterogeneous processing using both programmable logic and embedded processors. The PYNQ-Z1 provides a lower-resource FPGA platform intended for rapid prototyping through overlay-based hardware acceleration and Python-controlled software execution. Together, these boards provide different resource capacities and deployment interfaces, allowing the agent to compare implementation feasibility and performance across two FPGA platforms.

Communication between the orchestration node and the FPGA platforms combines local and network-based interfaces, reflecting the different access mechanisms exposed by the two boards. In the experimental setup, the PYNQ-Z1 was accessed through a USB/UART console, while the ZCU104 was accessed over Ethernet using SSH and file-transfer tools. This allows the proposed system to exercise the same orchestration workflow across different physical and network access mechanisms while abstracting board-specific connection details from the user.

Table 1 summarises the concrete access mechanisms employed throughout the experimental study. The orchestration workflow

Table 1: Device access methods used by the agent for the two FPGA platforms.

Device	Access method and capabilities
PYNQ-Z1	USB/UART access using `picocom` at 115200 baud, providing local shell access, board identification, file transfer, and deployment support.
ZCU104	Ethernet access using `ssh` and `scp`/`rsync`, providing Linux shell access, file transfer, and FPGA overlay and training execution.

treats the communication medium and access protocol as target-specific connection details rather than hard-coding a single access method. This allows the agent to interact with each FPGA platform through the interface it exposes while preserving a common deployment workflow for device access, file transfer, execution, and runtime measurement.

The FPGA software stack combines Vivado/Vitis HLS, synthesis and implementation, bitstream generation, overlay management, board drivers, Python control code, DMA support, and benchmarking scripts. The agent coordinates hardware artefacts such as bitstreams and hardware metadata with host-side control software, test data, and measurement logs.

There is no universally best FPGA target or implementation flow. A resource-constrained board may favour a compact LUT-based design, while a larger SoC FPGA can support greater parallelism and potentially higher throughput. The agent's role is therefore to jointly select the FPGA target, hardware-generation flow, quantisation strategy, interface structure, and deployment strategy under the measured constraints of the available devices.

Within this testbed, the same methodology can be exercised across several scenarios: discovering which boards are connected, determining how each can be accessed, profiling available resources, generating a candidate implementation, deploying it, and measuring the result. The measured metrics then decide whether the current deployment is acceptable, or whether the agent should reconfigure the hardware, change the quantisation strategy, regenerate the implementation, retrain the model, choose a different board, or ask the user to revise the objective.

4 Case Study: HVAC Energy Prediction

This section evaluates the proposed system through an HVAC energy-prediction case study involving two FPGA deployment targets and two hardware-generation flows. The evaluation examines whether the agent can: (i) discover and access the available edge platforms; (ii) generate feasible target-specific implementations; (iii) deploy and validate the generated artefacts on physical devices; (iv) explore implementation alternatives under accuracy, latency, throughput, and resource constraints; and (v) complete the end-to-end engineering workflow from device discovery to physical deployment and validation.

The user can express their requirements as a prompt – such as: generate a low-latency and high-throughput neural-network implementation for the HVAC dataset, compare suitable hls4ml and custom-HLS deployment options across the available FPGA boards, execute the selected candidates on physical devices, and report

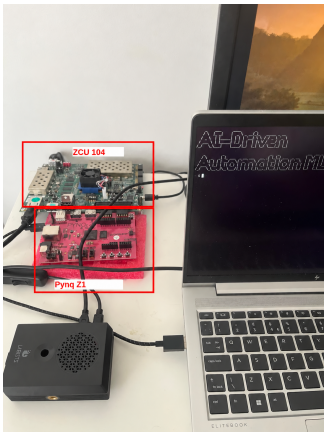


Figure 2: Experimental FPGA testbed comprising a host workstation connected to a ZCU104 SoC FPGA and a PYNQ-Z1 FPGA.

Table 2: HVAC energy-prediction workload used in the case study.

Item	Description
Dataset	HVAC dataset
Features	Occupancy, Rel_Hum, Room_Temp, Air_Flow_Rat, Air_Temp
Target	Elec_Cons
Task	Supervised regression
Test samples	3,464
Objective	Low latency and high throughput with acceptable prediction accuracy

measured accuracy and performance. The regression workload uses five input features (occupancy, relative humidity, room temperature, air-flow rate, and air temperature) to predict electrical consumption. Table 2 summarises the workload.

4.1 Agent-Driven Implementation Flow

The agent begins by discovering the available FPGA devices and selecting an access method for each. The ZCU104 was reached through Ethernet for file transfer and remote execution, while the PYNQ-Z1 was controlled through the USB/UART and PYNQ overlay workflow. Figure 3 summarises the deployment paths to the two FPGA targets. The available resources and connection medium at this stage determine which implementation is feasible and how the generated files can later be deployed, so the agent does not assume that two boards sharing a login prompt or IP address are the same target. Board identity was verified against the Linux device tree before deployment.

For FPGA targets, the agent generated two hardware paths. The first uses hls4ml as an automated neural-network-to-HLS baseline, with streaming I/O, a latency-oriented strategy, and a reuse factor of one. The second is a custom Vitis HLS implementation in which the dense layers, quantisation, stream packing, and data movement shell are explicitly generated, using AXI4-Stream input/output, an

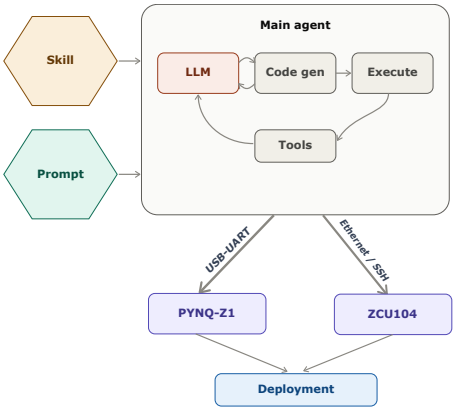


Figure 3: Agent-based deployment architecture in which the agent receives user objectives, generates FPGA and host-control artefacts, and deploys them to the PYNQ-Z1 and ZCU104 through board-specific connection media.

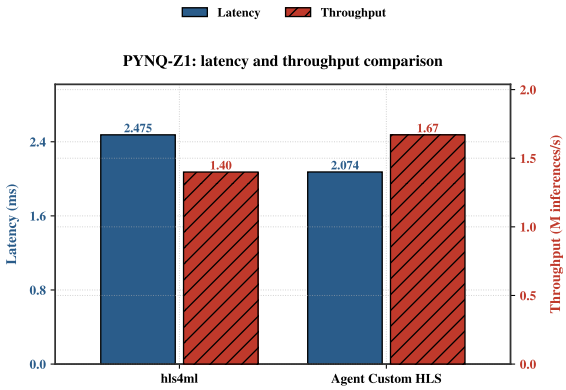


Figure 4: PYNQ-Z1 board-level latency and throughput for hls4ml and custom-HLS on the full test set.

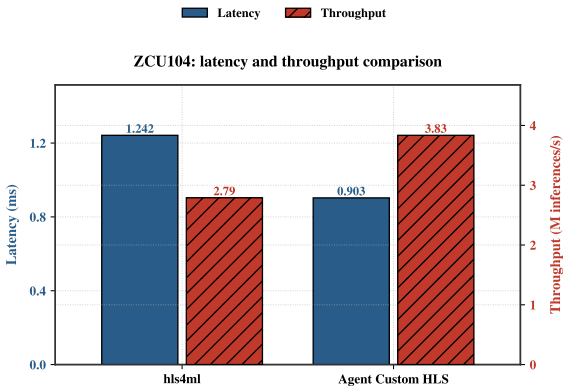


Figure 5: ZCU104 board-level latency and throughput for hls4ml and custom-HLS on the full test set.

AXI DMA interface, pipeline initiation interval one where applicable, array partitioning, and high parallelism in the dense-layer datapath. Both paths use the same 4-bit quantised neural-network architecture in the deployed comparison. Therefore, differences in measured performance primarily reflect the generated hardware architecture and board-integration flow rather than model differences.

The agent explored 8-bit, 6-bit, and 4-bit quantisation, summarised in Table 3. The optimised 4-bit configuration was chosen for final deployment because the user objective prioritised latency and throughput and accepted a moderate reduction in regression accuracy relative to wider 8-bit arithmetic. Using this 4-bit representation, the final implementation maps the arithmetic to LUT resources in the FPGA fabric and uses no DSP blocks. Some earlier 4-bit evaluations still report DSP usage, since the custom-HLS generator applied Vitis HLS binding pragmas to force selected multiply–accumulate operations into DSP blocks, as a test of whether explicit DSP binding would help; those DSP-forced variants were not selected for the final deployment. DSP blocks are designed for wider arithmetic than the 4-bit operations used here, making LUT-based multiplication more resource-efficient for the selected configuration; similar behaviour has been reported for compact 4-bit FPGA inference designs [7]. The generated overlays include the bitstream, hardware handoff metadata, AXI DMA integration, and Python driver scripts used to run the full test set.

4.2 Board Deployment and Runtime Measurement

The generated FPGA overlays were deployed using the PYNQ overlay model. Each bitstream was loaded with its corresponding hardware metadata file, and the same `X_test.npy` and `y_test.npy` files were used for both flows. The host-side driver allocated DMA buffers, streamed all 3, 464 test samples through the accelerator, collected the predicted output vector, converted it back to the original target scale, and computed R^2, Root-Mean-Square Error (RMSE), Mean Absolute Error (MAE), latency, and throughput.

Table 4 reports the measured board-level results, which include the PYNQ Python driver and DMA transfer overhead for the full test vector. On the PYNQ-Z1, the custom-HLS overlay completes the full test set in 2.074 ms against 2.475 ms for hls4ml, a throughput gain of roughly 19.4%. On the ZCU104, custom-HLS completes the same test set in 0.903 ms against 1.242 ms for hls4ml, a gain of roughly 37.5%. Both implementations reach the same measured regression accuracy on each board. Figures 4 and 5 show the combined latency and throughput measurements for each FPGA, and Figures 6 and 7 compare electrical-consumption traces from the Python reference data, hls4ml FPGA output, and custom-HLS FPGA output.

4.3 On-Device FPGA Training

The final experiment extends the workflow from inference-only acceleration to on-device training on the ZCU104 to verify that the generated FPGA design performs the training updates itself in programmable logic rather than evaluating a fixed model. The accelerator uses the same HVAC regression task and input features as before, but the hardware function now takes training samples,

Table 3: FPGA synthesis sweep for hls4ml and custom-HLS candidates. The table reports model accuracy and post-synthesis resource utilisation for each explored configuration. Fit indicates whether the design fits the target FPGA; Py. and HLS denote Python and HLS evaluation, respectively.

Board	Flow	Variant	Bits	Arch.	Fit	Py. R^2	HLS R^2	LUT	FF	DSP	BRAM18K
PYNQ-Z1	hls4ml	balanced	8	32–16–8	–	0.976488	0.976706	89,106	19,939	384	0
PYNQ-Z1	custom-HLS	balanced	8	32–16–8	–	0.976488	0.976706	101,024	21,564	103	0
ZCU104	hls4ml	balanced	8	32–16–8	✓	0.976488	0.976706	88,990	15,571	384	0
ZCU104	custom-HLS	balanced	8	32–16–8	✓	0.976488	0.976706	89,279	19,626	409	0
PYNQ-Z1	hls4ml	balanced	6	32–16–8	–	0.976248	0.965346	69,777	15,584	208	0
PYNQ-Z1	custom-HLS	balanced	6	32–16–8	–	0.976248	0.965346	79,783	17,460	61	0
ZCU104	hls4ml	balanced	6	32–16–8	✓	0.976248	0.965346	68,033	12,023	208	0
ZCU104	custom-HLS	balanced	6	32–16–8	✓	0.976248	0.965346	72,118	15,948	248	0
PYNQ-Z1	hls4ml	optimised	4	32–16–8	✓	0.955371	0.942238	28,918	7,583	0	0
PYNQ-Z1	custom-HLS	optimised	4	32–16–8	✓	0.955371	0.942238	30,761	9,239	0	0
ZCU104	hls4ml	optimised	4	32–16–8	✓	0.955371	0.942238	28,957	6,383	0	0
ZCU104	custom-HLS	optimised	4	32–16–8	✓	0.955371	0.942238	30,614	8,046	0	0
ZCU104	custom-HLS	max-capacity	8	44–24–12	✓	0.976125	0.976122	174,794	34,038	622	0
ZCU104	custom-HLS	max-capacity	6	44–24–12	✓	0.978115	0.957752	134,369	26,598	376	0
ZCU104	custom-HLS	max-capacity	4	44–24–12	✓	0.902669	0.885195	47,601	12,739	80	0
PYNQ-Z1	custom-HLS	PYNQ-fit	8	24–12–6	–	0.974193	0.974133	63,762	14,805	88	0
PYNQ-Z1	custom-HLS	PYNQ-fit	6	24–12–6	✓	0.969625	0.958323	49,757	11,944	63	0
PYNQ-Z1	custom-HLS	PYNQ-fit	4	24–12–6	✓	0.630500	0.638286	25,348	7,914	16	0

Table 4: Measured board-level FPGA inference results on the full 3,464-sample test set using the optimised 4-bit configuration.

Board	Flow	R^2	RMSE	MAE	Latency (ms)	Throughput (inf./s)
PYNQ-Z1	hls4ml	0.942238	0.004839	0.002006	2.475	1,399,316
PYNQ-Z1	custom	0.942238	0.004839	0.002006	2.074	1,670,480
ZCU104	hls4ml	0.942238	0.004839	0.002006	1.242	2,789,859
ZCU104	custom	0.942238	0.004839	0.002006	0.903	3,834,998

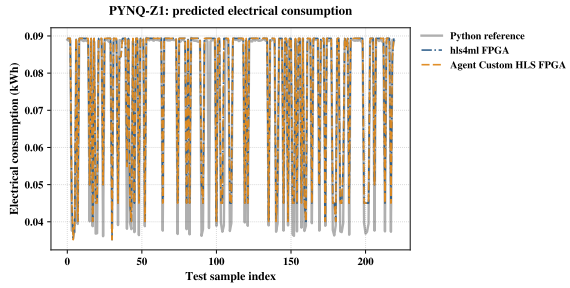


Figure 6: PYNQ-Z1 electrical-consumption prediction trace for the deployed 4-bit FPGA model. The vertical axis reports electrical consumption in kWh and the horizontal axis reports the test-sample index.

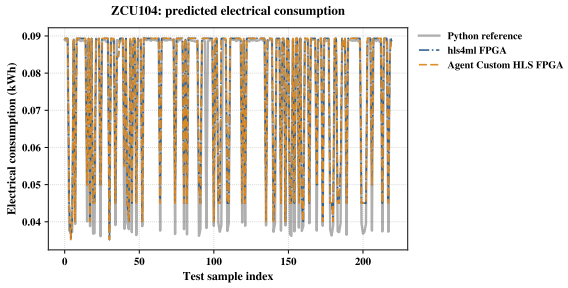


Figure 7: ZCU104 electrical-consumption prediction trace for the deployed 4-bit FPGA model. The vertical axis reports electrical consumption in kWh and the horizontal axis reports the test-sample index.

target values, initial weights, bias vectors, a learning rate, an epoch count, and output buffers for the updated weights and epoch loss.

The network is a compact multilayer perceptron model with architecture 5–32–16–8–1. The five inputs are occupancy, relative humidity, room temperature, air-flow rate, and air temperature; the output is predicted electrical consumption. Training uses mini-batch gradient descent with a batch size of 32 over 32 epochs, with each epoch processing 512 training samples. For each mini-batch, the FPGA performs the forward pass, computes the Mean Squared Error (MSE) derivative, backpropagates the errors, accumulates gradients, and updates the weights and biases. The weights remain in local memory during training and are written back to DDR once training completes, allowing the host processor to evaluate the trained model on the validation set.

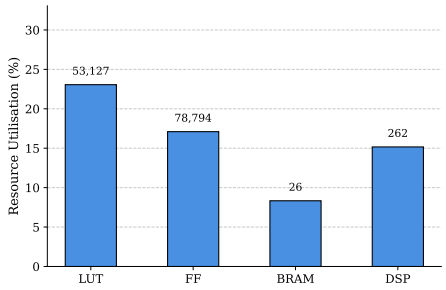


Figure 8: ZCU104 training accelerator resource utilisation showing percentage of total board resources and absolute values.

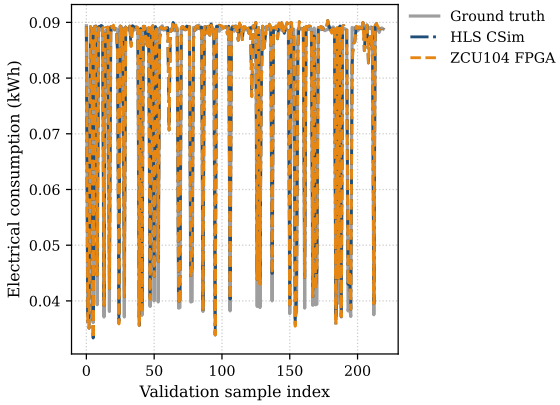


Figure 9: Validation prediction trace after on-device FPGA training. The vertical axis reports real electrical consumption in kWh and the horizontal axis reports the validation-sample index.

The HLS design favours local computation and predictable memory traffic. The top function exposes AXI4-Lite registers for control and eight AXI memory-mapped master interfaces for the data, labels, weights, status, and epoch-loss buffers. Weights and biases load from DDR into local arrays at the start of execution. The dense-layer weights, activations, deltas, and gradient buffers are partitioned so the inner-product and gradient loops can run in parallel. Each layer's forward function is synthesised separately, and the weight-update stage is pipelined so it can stream through local arrays at an initiation interval of one wherever data dependencies allow. DDR traffic is consequently limited to loading the initial tensors, reading training data and labels, writing the final weights, and writing the per-epoch loss trace, rather than reading and writing weights over AXI for every sample.

Training uses fixed-point arithmetic rather than floating point. Input features and labels use `ap_fixed<16,6>`, weights and biases use `ap_fixed<20,8>`, and accumulators use `ap_fixed<32,12>`. Activations and labels are kept narrow to save logic, while weights and gradient accumulators keep enough fractional range for stable Stochastic Gradient Descent (SGD) updates, since small gradient changes must still move the stored weights. This mixed-precision scheme is used throughout, in place of a single narrow inference-style format.

Three pieces of evidence support the validity of the training run. The FPGA reports the number of samples and epochs processed; the ZCU104 run processed 16, 384 samples, matching 512 samples times 32 epochs. The training loop writes loss values to the `epoch_loss` buffer, which fall from 0.950891 to 0.061243 over the run. The host also reads the updated weights back from DDR and recomputes the validation metrics from them: validation R^2 rises from -0.002568 before training to 0.957869 after, showing that the weights are genuinely updated rather than fixed.

Figure 8 reports the routed ZCU104 resource use of the training overlay. The design fits comfortably on the ZCU104 and meets timing with +0.404 ns setup slack, while using DSP blocks for the fixed-point training arithmetic and AXI-connected compute shell.

5 Results and Discussion

This case study demonstrates the full agentic workflow for FPGA synthesis and deployment. Starting from a natural-language goal, the agent identified the available FPGA devices, determined how to access them, generated hls4ml and custom-HLS implementations, tested different quantisation settings, produced FPGA overlays, transferred the deployment files, executed the models on physical hardware, and collected accuracy and runtime results.

The on-device training experiment extends this workflow beyond fixed inference. The ZCU104 training accelerator processed 16, 384 training samples across 32 epochs, reducing the FPGA-reported training MSE from 0.950891 to 0.061243 and raising validation R^2 from -0.002568 before training to 0.957869 after. The HLS C-simulation and physical FPGA run produced the same final loss and validation R^2, while the Python floating-point reference reached a close R^2 of 0.962255. These results confirm that the accelerator performs mini-batch SGD updates in hardware, and that the returned weights are usable for post-training validation on the host processor.

Training also poses a different optimisation problem from inference. In the inference designs, the agent could push arithmetic precision down to 4-bit aggressively because the weights are fixed after training and the objective prioritised latency and throughput. For training, it used mixed fixed-point arithmetic instead: `ap_fixed<16,6>` for inputs and labels, `ap_fixed<20,8>` for weights and biases, and `ap_fixed<32,12>` for accumulators, since gradient updates need enough fractional resolution to converge. The routed ZCU104 overlay met timing with 0.404 ns setup slack, using 53, 127 LUTs, 78, 794 registers, 26 BRAM tiles, and 262 DSP blocks. The design fits comfortably within the ZCU104's available resources while preserving training convergence.

The quantisation sweep in Table 3 shows that the highest-accuracy bit width in Python is not automatically the best choice for hardware. The 8-bit and 6-bit designs retain higher accuracy, with HLS R^2 above 0.96 for the balanced 32–16–8 architecture, but they either fail to fit on the PYNQ-Z1 under the fully parallel hls4ml and custom-HLS settings, or consume a much larger share of the available logic and DSP resources. The user objective here prioritised low latency and high throughput given acceptable accuracy, which

shifts the target from "highest possible accuracy" to "best hardware performance under an accuracy constraint".

The 4-bit optimised configuration is the only precision setting that meets this goal across both FPGA boards and implementation flows. It fits the smaller PYNQ-Z1 and also maps well onto the larger ZCU104. Its HLS R^2 of 0.942238 is lower than the 8-bit and 6-bit designs, but the drop is small for this regression task and an acceptable trade-off for a smaller, faster design. Board measurements confirm that both 4-bit FPGA implementations give the same test-set accuracy, so the comparison between the two FPGA flows reflects only the hardware flow, not model quality.

The 4-bit result also carries a useful hardware lesson. In the final, optimised configuration, all arithmetic maps to LUTs and no DSP blocks are used, which makes sense, since 4-bit multipliers are small enough to implement efficiently in LUTs, while DSP blocks are built for wider arithmetic. Earlier 4-bit variants that did use DSP blocks were kept in the sweep as a deliberate test of DSP-binding pragmas, checking whether forcing small multiply-accumulate operations into DSPs would help; the final design avoids DSPs where LUTs already suffice.

The board-level results in Table 4 show the custom-HLS overlay outperforming the hls4ml overlay on both FPGA boards for identical R^2, RMSE, and MAE. On the PYNQ-Z1, custom-HLS processes the full 3,464-sample test set in 2.074 ms compared with 2.475 ms for hls4ml, increasing throughput from 1.399 million to 1.670 million inferences per second. On the ZCU104, custom-HLS reduces runtime from 1.242 ms to 0.903 ms, increasing throughput from 2.790 million to 3.835 million inferences per second. Since the test set and the neural network architecture are identical between the two flows, the performance difference primarily reflects the agent-generated hardware architecture and implementation flow.

The speed-up mainly reflects how specifically the custom design targets this regression task. With only five input features and a single output value, the agent can build a compact, streaming datapath rather than a general neural-network wrapper. The custom-HLS flow uses AXI4-Stream interfaces, DMA transfers, explicit stream packing, array partitioning, and an initiation interval of one in the core datapath, so the accelerator accepts a new sample every clock cycle once the pipeline is full. hls4ml remains a strong baseline, since it already supports neural-network conversion, streaming input/output, and latency-focused synthesis, but it keeps a more general layer structure and conversion flow, which is useful for automation but adds scheduling and interface overhead relative to a design built for one specific workload.

This does not make hls4ml a weak tool; it offers a reliable, well-structured path from trained neural networks to HLS and is particularly useful when a design needs to stay close to the original model-development environment. Rather, the result illustrates the value of letting the agent specialise the hardware once a candidate model and interface are chosen: it can remove unneeded generality, fix the datapath to the known data shapes, and simplify the streaming interface. Here, that specialisation is enough to outperform the hls4ml baseline at board level.

An FPGA should not be assumed to be the best deployment target for every edge-ML workload. For small regression models such as the one considered here, software overhead, data movement, and batching strategy can substantially influence end-to-end performance. Although the quantitative evaluation in this work focuses on FPGA platforms, a broader deployment study should also compare against optimised processor-based implementations to determine when FPGA acceleration provides a meaningful advantage.

Within the FPGA platforms evaluated here, the custom ZCU104 overlay achieves the highest throughput, reaching 3.835 million inferences per second on the full test set. Its fixed streaming datapath also makes it suitable for workloads requiring sustained hardware-level processing and integration with other programmable-logic components.

The ZCU104 result also shows why board resources matter. The larger device provides substantially more LUTs, flip-flops, DSP blocks, BRAM capacity, and routing resources than the PYNQ-Z1, enabling higher-throughput overlays and more ambitious custom-HLS designs. The smaller PYNQ-Z1 remains useful for evaluating whether a design can fit within tighter FPGA resource constraints. Among the two FPGA platforms evaluated, the ZCU104 is therefore the stronger choice when the objective is to maximise throughput and resource availability is not the limiting factor. This reinforces the central argument of the work: the agent should jointly consider model quality, quantisation, resource fit, clock speed, interface requirements, target-device capabilities, and measured board-level behaviour when selecting and refining an implementation, rather than optimising for accuracy alone.

6 Conclusion

This paper presented an agentic engineering system for automating and iteratively refining the deployment of ML models and workflows across FPGA-based edge hardware. The work addressed the engineering challenge of coordinating deployment decisions across hardware discovery, resource profiling, target selection, model optimisation, implementation generation, toolchain configuration, physical deployment, and runtime validation. Rather than treating these activities as isolated stages, the proposed system coordinated them through a closed-loop workflow in which measured execution results and deployment failures informed subsequent decisions.

The system was evaluated using an HVAC energy-prediction workload on a physical testbed comprising a PYNQ-Z1 FPGA and a ZCU104 SoC FPGA. The agent generated and deployed both hls4ml-based and custom-HLS FPGA implementations, explored different quantisation settings and hardware-generation flows, and collected measurements from the two physical FPGA boards. Notably, the custom-HLS implementation achieved up to 3.83 million inferences per second on the ZCU104, providing a 37.5% throughput improvement over the hls4ml implementation while maintaining the same deployed regression accuracy. The system also generated and deployed an on-device FPGA training accelerator.

Overall, the results show that agentic AI can automate a substantial part of the FPGA-based edge-ML deployment process. Future work will extend the system to additional hardware targets and ML workloads, while improving automated target selection, failure handling, and design-space exploration.

Acknowledgments

This work was partially supported by the EPSRC-funded "UK National Edge AI Hu" (https://edgeaihub.co.uk/).

References

[1] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. 2021. MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers. *Proceedings of Machine Learning and Systems* 3 (2021), 517–532.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[3] Robert David, Jared Duke, Advait Jain, Vijay Janapa Reddi, Nat Jeffries, Jian Li, Nick Kreeger, Ian Nappier, Meghna Natraj, Tiezhen Wang, Pete Warden, and Rocky Rhodes. 2021. TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 800–811. https://proceedings.mlsys.org/paper_files/paper/2021/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf

[4] MG Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. 2021. Machine Learning at the Network Edge: A Survey. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–37.

[5] Jan-Frederik Schulte, Benjamin Ramhorst, Chang Sun, Jovan Mitrevski, Nicolò Ghielmetti, Enrico Lupi, Dimitrios Danopoulos, Vladimir Lončar, Javier Duarte, David Burnette, et al. 2026. hls4ml: A Flexible, Open Source Platform for Deep Learning Acceleration on Reconfigurable Hardware. *ACM Transactions on Reconfigurable Technology and Systems* 19, 2 (2026), 1–35.

[6] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–74.

[7] Bingrui Zhao, Yaonan Wang, Hui Zhang, Jinzhou Zhang, Yurong Chen, and Yimin Yang. 2023. 4-bit CNN Quantization Method With Compact LUT-Based Multiplier Implementation on FPGA. *IEEE Transactions on Instrumentation and Measurement* 72 (2023), 1–10. doi:10.1109/TIM.2023.3324357

[8] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. 2019. Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing. *Proc. IEEE* 107, 8 (2019), 1738–1762.