

Article

Integrating AutoML and FPGA Deployment: A Pipeline for Model Selection and Hardware-Aware Implementation

Yrskeldi Siddi , Giorgio Delzanno and Daniele D'Agostino * 

Department of Informatics, Bioengineering, Robotics and Systems Engineering (DIBRIS), Università degli Studi di Genova, 16146 Genova, Italy; s6541657@studenti.unige.it (Y.S.); giorgio.delzanno@unige.it (G.D.)

* Correspondence: daniele.dagostino@unige.it

Abstract

The increasing demand for real-time, energy-efficient machine learning (ML) models in edge and embedded scenarios highlights the limitations of traditional CPU-based inference pipelines. This work presents a pipeline that integrates Automated Machine Learning (AutoML) with High-Level Synthesis (HLS) to enable efficient deployment of ML predictors on Field-Programmable Gate Arrays (FPGAs). Using Auto-Weka for automated algorithm selection and Bayesian hyperparameter optimization, followed by model reimplementing and parameter extraction using scikit-learn and synthesis through the AMD/Xilinx Vitis HLS toolchain, the proposed workflow combines data-driven model exploration with hardware-oriented implementation. The pipeline adopts a hybrid, human-in-the-loop approach, reflecting current practical constraints in bridging heterogeneous software and hardware environments. Experimental results show up to a 9% latency reduction compared to CPU-based inference, more than a 7× throughput improvement through parallel FPGA instantiation, and more than an order of magnitude improvement in energy efficiency when evaluated in terms of energy per inference. The proposed methodology provides a reproducible and extensible workflow for integrating AutoML-based model discovery with FPGA deployment, while highlighting both the benefits of hardware acceleration and the remaining challenges in AutoML–hardware integration.

Keywords: AutoML; High-Level Synthesis; FPGA acceleration; edge AI; Multinomial Logistic Regression; embedded AI; hardware-aware ML

1. Introduction

Artificial Intelligence is rapidly shifting toward edge-centric computation [1], where tasks must be performed close to the data source with tight constraints on latency, energy consumption, privacy, and real-time responsiveness [2]. Wearables, IoT platforms, industrial automation, mobile robotics, and smart health monitoring all require lightweight, deployable, energy-efficient machine learning (ML) models [3].

Traditional machine learning pipelines, while highly effective in unconstrained environments, present several structural limitations when applied to real-world edge and embedded scenarios [4]. These limitations arise not from deficiencies in learning algorithms themselves, but from the complexity of the surrounding engineering processes required to design, optimize, and deploy such models efficiently [5].

A first major challenge concerns the complexity of model engineering. Selecting an appropriate learning algorithm and tuning its hyperparameters often requires substantial expertise both in the applicative domain and in ML, together with extensive



Academic Editor: Giovanni Diraco

Received: 15 May 2026

Revised: 3 July 2026

Accepted: 7 July 2026

Published: 10 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

experimentation. For non-expert practitioners, this process is particularly burdensome, as it involves navigating a vast design space with little guidance and high sensitivity to configuration choices [6,7]. As a result, model development frequently relies on manual trial-and-error procedures, which are time-consuming, difficult to reproduce, and prone to suboptimal outcomes.

The second challenge is tightly coupled with the first, and emerges at the deployment stage on resource-constrained hardware platforms. Although CPUs and GPUs dominate traditional machine learning infrastructures, they are often ill-suited for edge environments characterized by limited power budgets, real-time constraints, and thermal limitations [8,9]. GPU-based acceleration, in particular, provides high throughput at the cost of significant energy consumption, making it unsuitable for battery-powered devices or always-on systems. Similarly, CPU-based inference suffers from limited parallelism and unpredictable latency, which can be critical drawbacks in time-sensitive applications. This is the reason why FPGA-based solutions are even more considered [10].

The major barrier preventing the wide adoption of FPGA-based solutions lies in the complexity of hardware design. Conventional FPGA development demands expertise in register-transfer level (RTL) design, timing closure, resource mapping, and digital logic optimization. This steep learning curve effectively separates machine learning practitioners from hardware acceleration opportunities, reinforcing a divide between algorithm designers and hardware engineers. Together, these challenges highlight a fundamental disconnect between modern machine learning workflows and practical deployment on edge devices. Addressing this disconnect requires not only advances in learning algorithms, but also integrated methodologies that reduce engineering complexity across the entire pipeline—from model selection to hardware execution.

In this paper we investigate the possibility to target these issues by introducing an integrated workflow that jointly addresses model engineering, deployment efficiency, and hardware accessibility. Rather than managing model selection, training, and deployment as isolated stages, the proposed framework aims to integrate them into a coherent workflow tailored for edge-oriented artificial intelligence.

This work makes several original contributions to the field of edge-oriented machine learning systems. First, the paper introduces a novel pipeline that integrates Automated Machine Learning (AutoML) with High-Level Synthesis (HLS) to enable efficient machine learning acceleration on FPGA platforms. Unlike traditional workflows, which treat model selection and hardware deployment as largely disconnected phases, the proposed approach establishes a tight integration between these stages, improving design coherence and reducing engineering complexity. It is worth noting that the proposed workflow does not aim to fully automate the entire machine learning-to-hardware design process. Instead, it adopts a hybrid, human-in-the-loop approach, where automated model selection is complemented by expert-driven integration into the hardware design flow. This reflects current practical constraints, as seamless interoperability between AutoML frameworks and hardware synthesis tools is not yet fully supported.

Second, the pipeline is validated through a real-world case study based on a physiological sensor dataset comprising more than 18,000 real observations collected from wearable devices in the project S.F.I.D.A. (The Smart Frail Individuals Data Analysis (S.F.I.D.A.) Project: Regional Programme 2021–2027, Priority 1—Development and consolidation of the competitiveness and attractiveness of the Ligurian system, with particular reference to innovation, digitalization, and strengthening; Specific Objective 1.1: Develop and strengthen research and innovation capabilities and the adoption of advanced technologies; Action 1.1.1—Call for Proposals: “Support for the implementation of research and experimental development projects for companies associated with Research and Innovation Clus-

ters”, Liguria Region, Italy (Website <https://comftech.com/it/progetto/progetto-sfida>, accessed on 7 July 2026). The considered applicative scenario demonstrates the practical relevance of the proposed methodology in realistic edge AI contexts, such as health monitoring and sensor-driven environments, where constraints on latency, energy consumption, and reliability are particularly stringent.

Third, an experimental evaluation is conducted to assess the effectiveness of the proposed pipeline. Performance is analyzed across multiple dimensions, including inference latency, throughput, FPGA resource utilization, power consumption, and the impact of architectural optimizations introduced during HLS. This multidimensional evaluation provides a comprehensive picture of the trade-offs involved in deploying machine learning models on reconfigurable hardware.

Finally, beyond the specific dataset and models considered, the paper introduces a methodology that can potentially be extended to a broader range of applications. The proposed pipeline is not inherently tied to a particular application or learning task, although the current validation focuses on a specific use case. As such, it provides a reusable blueprint for bridging AutoML-based model discovery and hardware-efficient deployment in edge AI applications.

The remainder of this paper is structured as follows: Section 2 reviews the related literature on AutoML and its integration with FPGA-based systems. Section 3 describes the physiological monitoring application scenario and the dataset used for validation. Section 4 details the proposed end-to-end pipeline, from model selection to HLS-based hardware implementation. Section 5 presents the experimental results, including performance and energy efficiency analysis. Section 6 discusses the trade-offs and limitations of the approach, and finally, Section 7 concludes the paper and outlines future research directions.

2. Related Works

2.1. AutoML

The term Automated Machine Learning (AutoML) was introduced in 2015 in the context of the ChaLearn competition [11], with the objective of defining tools and methodologies capable of automating substantial portions of the machine learning workflow [12]. The motivation behind AutoML stems from the intrinsic complexity of designing effective learning models, a process that traditionally requires advanced expertise in machine learning algorithms and in the specific application domain as well [13]. A major issue today is the insufficient number of ML experts. Therefore non-expert users often resort to heuristic algorithm selection, default hyperparameter settings, or ad hoc trial-and-error strategies. Alternatively, brute-force exploration techniques may be employed to search the space of possible configurations.

However, the combination of a large number of learning algorithms with extensive hyperparameter spaces quickly made these approaches methodologically inefficient and possibly computationally impractical [14]. As a consequence, despite the common perception of machine learning as a paradigm that reduces the need for explicit programming, effective model development still demands a considerable investment of time, manual effort, and iterative experimentation.

One of the first tools for AutoML is represented by Auto-Weka [15]. Developed in 2013, it introduced a system that automates the selection and configuration of machine learning models in the WEKA platform. The goal is to reduce the need for manual intervention and improve model quality through automatic optimization.

WEKA (Waikato Environment for Knowledge Analysis) is an open-source platform for data mining and machine learning [16]. It offers a vast collection of algorithms for

classification, regression, clustering and feature selection and is used primarily in academic settings for experimentation and teaching. Auto-WEKA was designed with the explicit goal of exploring a very large configuration space composed of (1) the number of algorithms available, (2) multiple data preprocessing operators, and (3) extensive hyperparameter ranges for each component. From a conceptual standpoint, Auto-WEKA performs a joint optimization over algorithms, preprocessing steps, and hyperparameters, resulting in an unusually broad search space for an AutoML system. This makes Auto-WEKA particularly attractive in research-oriented settings, where algorithm-agnostic evaluations and methodological comparisons are required.

However, such generality comes at the cost of high computational complexity [15,17]. The inclusion of a large number of heterogeneous learners—many of which are rarely competitive on modern datasets—significantly enlarges the search space and can lead to inefficient exploration [18,19]. As a consequence, while Auto-WEKA is highly expressive, it often struggles to scale to large datasets or limited time budgets.

In contrast, most modern AutoML frameworks (<https://www.automl.org/automl/>, accessed on 7 July 2026) adopt a fundamentally different perspective. Rather than aiming for exhaustive coverage of all possible machine learning algorithms, these systems deliberately restrict the set of candidate models to a relatively small subset of algorithms that have demonstrated strong empirical performance across a wide range of tasks [20,21]. This design choice reflects a shift from algorithmic completeness to practical effectiveness. Modern AutoML tools prioritize robustness across datasets, computational efficiency, and reproducibility of results. Focusing on a smaller number of well-performing algorithms often leads to better results, especially when combined with ensemble methods and meta-learning strategies.

Auto-sklearn [22] is conceptually the closest alternative to Auto-WEKA. It performs combined algorithm selection and hyperparameter optimization using models from the scikit-learn ecosystem. While it still supports a variety of classifiers and regressors, the number of considered algorithms is significantly smaller than in Auto-WEKA. This restriction is intentional and improves convergence speed, ensemble stability, and overall predictive performance. Thus, although Auto-sklearn is less complete in terms of the raw number of models, its goal is to offer a more balanced trade-off between generality and practical usability.

TPOT [23] focuses on the automatic construction of machine learning pipelines through genetic programming. Its strength lies in the flexible combination of preprocessing steps and estimators, rather than in the sheer number of distinct learning algorithms.

Frameworks such as H2O AutoML [24] and AutoGluon [21] take an even more restrictive approach. They concentrate on a small number of high-performing model families, such as gradient boosting machines, random forests, and neural networks, often combined through stacking or ensemble learning. These tools make no attempt to be algorithm-complete in the sense of Auto-WEKA. Instead, they aim to maximize predictive performance under realistic computational constraints.

FLAML [25] explicitly optimizes for cost-effectiveness. Both the number of considered algorithms and the hyperparameter search strategies are constrained to minimize computation time while maintaining competitive accuracy. This makes FLAML particularly suitable for resource-limited environments, but also clearly less comprehensive than Auto-WEKA in terms of algorithm diversity.

2.2. AutoML and FPGA

The application of AutoML techniques to FPGA-based systems is gaining increasing attention in particular as a means to address the growing complexity of hardware–software co-design for machine learning accelerators.

A general-purpose AutoML-based co-design approach for FPGA inference is presented in [26]. In this work, the authors propose a Neural Architecture Search (NAS) framework targeting Multilayer Perceptron (MLP) networks, in which neural architecture parameters—such as depth, number of neurons, activation functions, and numerical precision—are jointly optimized with FPGA hardware parameters, including parallelism degree, pipelining strategies, resource utilization, and operating frequency. The resulting multi-objective optimization problem considers accuracy, latency, throughput, and hardware cost, producing Pareto-optimal solutions that are validated on real FPGA platforms.

A complementary, domain-specific perspective is proposed in [27], where AutoML is applied to the design of FPGA accelerators for Radio Machine Learning (RadioML) applications. Unlike the previous work, which focuses on MLP inference and design generality, this approach targets the stringent requirements of radio signal processing systems, such as real-time operation, low latency, and efficient resource usage. The proposed hardware-aware AutoML framework directly integrates hardware metrics into the optimization loop, jointly exploring machine learning parameters (e.g., network structure, parallelism, and quantization) and FPGA implementation choices. The framework enables the automatic generation of custom accelerators optimized for RadioML workloads rather than general-purpose inference.

Beyond these two representative contributions, several works have explored hardware-aware NAS and AutoML techniques for FPGA platforms. Multi-objective NAS approaches, such as APNAS [28], explicitly incorporate accuracy-performance trade-offs using hardware cost models, while more recent frameworks extend AutoML pipelines to include compression and quantization as first-class optimization variables [29]. In parallel, hardware-centric AutoML methods for mixed-precision and fixed-point quantization [30] further emphasize the importance of jointly optimizing numerical representation and hardware implementation, particularly in FPGA-based design flows.

From a broader perspective, the integration of AutoML and hardware-aware NAS techniques into FPGA-based design flows has been widely recognized as a key research direction. Comprehensive surveys highlight the limitations of manual and sequential design approaches and emphasize the importance of joint optimization of neural architectures, numerical precision, and hardware implementation parameters, particularly for resource-constrained and performance-critical platforms such as FPGAs [31]. These works position FPGA-oriented AutoML frameworks as a natural evolution of early AutoML systems, extending general-purpose techniques towards hardware-aware and application-driven co-design paradigms [32].

While the aforementioned approaches demonstrate the effectiveness of hardware-aware AutoML and NAS techniques, it is important to highlight a key distinction with respect to the methodology proposed in this work. Existing frameworks such as APNAS and CoDAC focus on integrating hardware constraints directly into the model design and optimization process, typically targeting neural network architectures and relying on unified, model-specific toolchains.

In contrast, the approach presented in this paper addresses a different and complementary challenge, namely the integration of general-purpose AutoML frameworks with FPGA-based deployment flows in heterogeneous environments. Rather than modifying the learning process or restricting the model space, the proposed pipeline focuses

on bridging the gap between software-centric model generation (e.g., Auto-Weka) and hardware-oriented implementation using HLS tools.

As a consequence, the two classes of approaches operate at different levels of the design stack and are not directly comparable in terms of experimental metrics. While hardware-aware AutoML frameworks optimize model architectures under hardware constraints, the proposed methodology emphasizes system-level integration, interoperability between tools, and practical deployment on real FPGA platforms.

3. Applicative Scenario

The S.F.I.D.A. (The Smart Frail Individuals Data Analysis) project (<https://comftech.com/it/progetto/progetto-sfida>, accessed on 7 July 2026) has been focused on developing a living lab to support the transfer of wearable technologies and AI-driven methods into the health sector. ComfTech (<https://comftech.com/>, accessed on 7 July 2026) is the lead partner of the project. The partners involved were University of Genoa, Department of Informatics, Bioengineering, Robotics and Systems Engineering (DIBRIS), Wondertech (<https://www.wondertechweb.com/>, accessed on 7 July 2026), and INSUDE (<https://www.insyde.eu/>, accessed on 7 July 2026). The purpose of the project is to use certified wearable devices to collect high-quality data for training complex systems through AI-based tools and validate solutions supporting citizens, including individuals in fragile conditions. The project had different work packages including the use of certified wearable devices, more specifically, the Howdy Senior kit (<https://comftech.com/en/project/howdy-senior/>, accessed on 7 July 2026) for collecting physiological data from frail users, with a particular focus on ECG, heart rate, and respiration during physical activity. Another task, strictly related to the present paper, was to develop a programmable hardware platform based on a next-generation board capable of running some of the developed algorithms directly on-device, enabling the creation of a new and more advanced generation of sensors. The pipeline proposed in this paper was applied and evaluated within this context.

Continuous observation of human motion and physiological signals plays a crucial role in both performance analysis and safety monitoring, particularly in sports and endurance-based activities.

Physiological measurements such as heart rate, respiration rate, and breath frequency provide valuable insights into an individual's physical condition and exertion level. When collected continuously, these signals enable a fine-grained analysis of athletic performance and recovery patterns. Moreover, in long-duration or high-intensity activities, real-time monitoring of physiological parameters can serve a preventive and safety-oriented function, allowing for the early detection of abnormal conditions or signs of physical discomfort and thus supporting timely intervention.

Within this context, a suitable machine learning model is required to satisfy two complementary requirements. At first, it must achieve high classification accuracy in identifying the type of physical activity performed by each subject from multivariate physiological data. On the other hand, the model must maintain limited computational complexity, as the target scenario involves real-time inference on continuous data streams acquired simultaneously from multiple individuals possibly on the field. Ensuring low latency and computational efficiency is therefore essential for scalability and for enabling the monitoring of a large number of users without compromising responsiveness or energy efficiency.

Data Format

The dataset used for the tests consists of real-time monitoring data collected from wearable sensors during physical activities, capturing both physiological and motion-related signals. This dataset provides a realistic and challenging testbed for evaluating the proposed pipeline, as it reflects the characteristics of data typically encountered in edge AI applications, including temporal continuity, inter-subject variability, and strict real-time constraints.

In detail, the dataset that has been collected so far contains 18,422 rows with the following columns:

- **Timestamp:** The date and time of the reading (in the format YYYY-MM-DD HH:MM:SS). Data is recorded at a frequency of one second.
- **Heart Rate (num):** The heart rate measured in beats per minute. Values range from 60 to 180 bpm.
- **Breath frequency (num):** The number of breaths taken per minute, ranging from 12 to 30 bpm.
- **Respiration (list):** The value of Respiration in a list format.
- **AccelerationX, AccelerationY, AccelerationZ:** Numerical values representing acceleration in three directions. Ranges are (−0.90, 0.44) X-axis, (−0.97, 1.08) Y-axis.
- **Position, phone longitude, phone latitude, phone altitude:** Numeric values representing the position of the device.
- **Ecg (list):** Electrocardiogram data (numeric values).
- **R2R:** Numeric values related to the Record-Record interval of the ECG.
- **Label:** Nominal value associated with each row (numerical values): 1—At rest; 2—Walking on flat ground; 3—Climbing stairs; 4—Walking downhill; 5—Walking uphill. The labels have been inserted manually.

4. Materials and Methods

Differently from hardware-aware AutoML and NAS-based co-design approaches, which integrate model exploration and hardware optimization within a unified automated loop, this work adopts a deployment-oriented perspective. In particular, it focuses on the practical integration of software-centric AutoML tools with standard FPGA design flows based on HLS. The overall pipeline architecture is illustrated in Figure 1.

The AutoML stage is implemented with Auto-Weka. As discussed before, it is the main system that attempts near-exhaustive algorithm selection. This level of completeness is advantageous from a methodological point of view [32]. By leveraging Bayesian optimization techniques, this stage replaces manual trial-and-error procedures with a principled and fully reproducible model search strategy. As a result, high-performing models can be identified directly from the data without requiring extensive domain expertise, thereby transforming model discovery into a data-driven and systematic process.

The pipeline moves to the Hardware-Aware C++ Implementation stage, where the optimal configuration is mapped into a C++ kernel tailored for synthesis. This intermediate representation constitutes a crucial transition layer, translating high-level machine learning abstractions into a form that can be efficiently optimized at the architectural level without compromising functional correctness. As we will discuss, this stage still requires a heavy human intervention.

The C++ kernel is subsequently processed using High-Level Synthesis (HLS) through AMD/Xilinx Vitis HLS, which automates the transformation from algorithmic code to a register-transfer level (RTL) hardware design. During this third stage, the HLS tool performs operation scheduling, pipelining, and resource allocation, mapping the inference computation onto the fine-grained parallel fabric of the target FPGA. By abstracting

low-level hardware design details, HLS significantly lowers the barrier to FPGA adoption while exposing the intrinsic performance benefits of reconfigurable logic.

Finally, the synthesized design is deployed onto the target FPGA platform, where pipelined and parallel execution enable low-latency and energy-efficient inference. The following subsections provide a more detailed discussion of these stages.

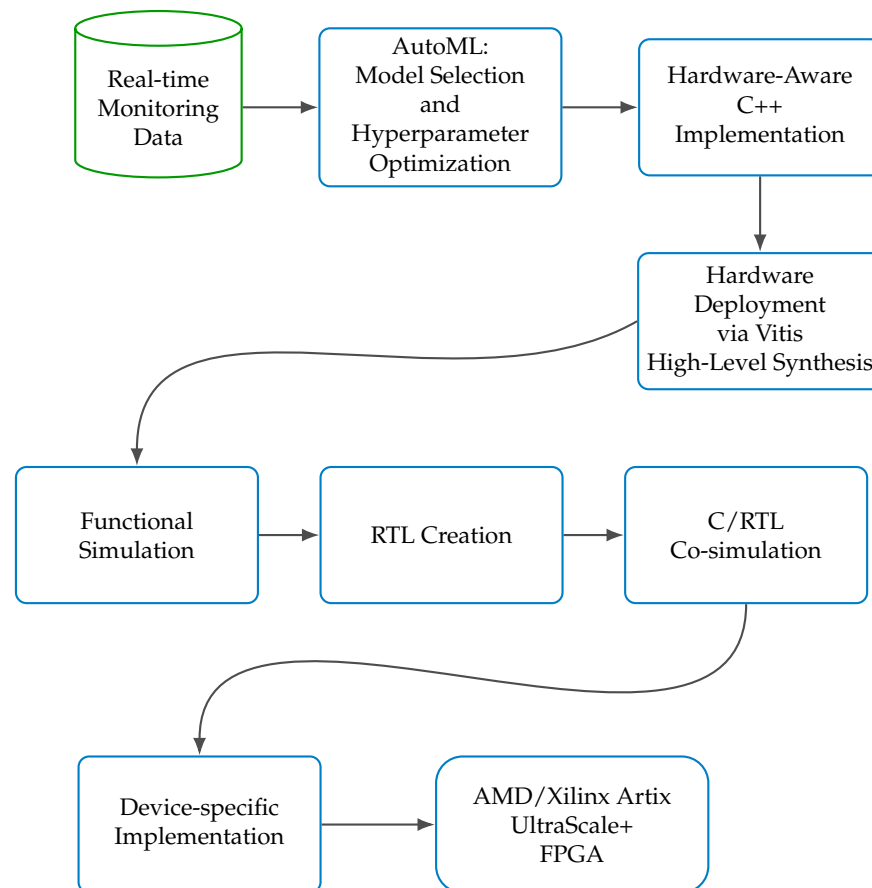


Figure 1. Overview of the (partially automated) AutoML-to-FPGA pipeline for deploying machine learning models on FPGA-based edge systems. AutoML is used for automated model selection and hyperparameter optimization. The selected model is then re-implemented in hardware-aware C++ and synthesized using the Vitis High-Level Synthesis toolchain, including functional simulation, RTL generation, co-simulation, and device-specific implementation on an AMD/Xilinx FPGA.

4.1. AutoML: Model Selection and Hyperparameter Optimization

To better understand the role and benefits of the AutoML stage, it is important to consider the complexity of the tasks it is designed to automate. A supervised machine learning workflow is typically structured into three main phases: data preparation, feature engineering, and model generation. While each stage introduces its own challenges, model generation is particularly sensitive to algorithmic and hyperparameter choices, which directly impact both predictive performance and computational cost.

Auto-Weka relies on Sequential Model-Based Algorithm Configuration (SMAC), a Bayesian optimization technique that employs random forest surrogate models to approximate the relationship between hyperparameter configurations and classification error. By leveraging an expected improvement acquisition criterion, SMAC balances exploration of unexplored regions with exploitation of promising configurations, allowing efficient navigation of the search space without exhaustive evaluation. Therefore Auto-Weka, thanks to Bayesian optimization, reduces the number of experiments needed to find optimal con-

figurations [14]. In several benchmarks, Auto-Weka has achieved results comparable to or superior to human experts [15].

In the proposed pipeline, Auto-Weka is employed exclusively for classification model selection and hyperparameter optimization. Feature engineering and data preprocessing are intentionally handled outside the AutoML loop in order to maintain full control over real-time constraints, streaming behavior, and system-level requirements. This design choice ensures that the resulting models are not only accurate, but also compatible with the architectural and temporal constraints imposed by FPGA-based edge deployment.

Auto-Weka defines both the search space and the computational budget according to user-specified parameters, including the set of candidate algorithms, hyperparameter ranges, time limits, and memory constraints. Given a set of parameters (see Figure 2), the optimization process starts by generating some random configurations. These initial configurations serve to bootstrap the surrogate model used during the subsequent optimization phase.

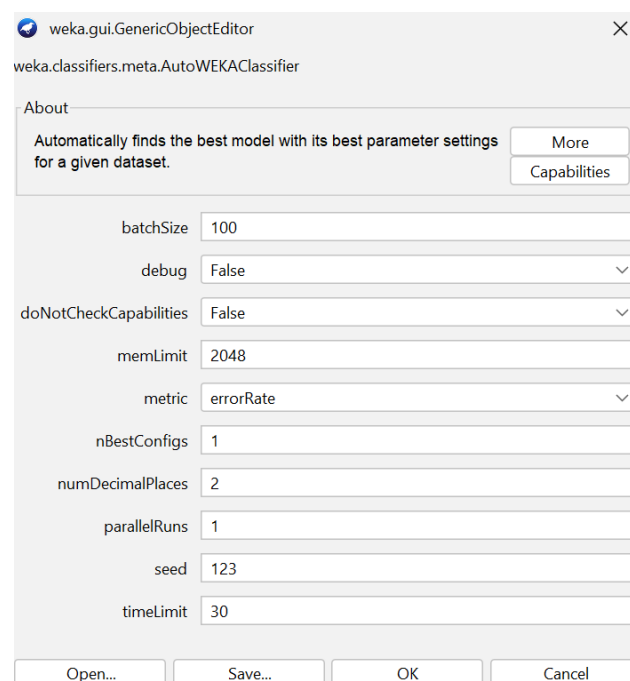


Figure 2. Graphical user interface (GUI) used to configure the Auto-Weka search parameters.

Regarding feature representation, the ECG and respiration signals were originally available as time-series (list-type) data and were therefore transformed into scalar features before model training. Specifically, values collected within each time window were aggregated through feature extraction procedures. For signals such as ECG, statistical descriptors (e.g., average values) were computed over the window, whereas for other physiological variables only the most recently updated value within the considered window was retained.

This pre-processing step produced a fixed-length feature vector, reformulating the task as a conventional supervised classification problem, where each feature vector is associated with a corresponding class label.

Each candidate configuration is evaluated within the AutoML process using cross-validation, with data partitions generated through random sample-level splits (80/20 train-validation). Moreover, ECG and respiration signals, originally represented as list-type features, were transformed into scalar inputs through statistical aggregation.

In the context of low-frequency physiological time-series data, the adopted sample-level data partitioning strategy may introduce temporal correlations between training and

validation samples. In particular, contiguous or near-contiguous observations originating from the same physiological or activity segment may be split across the two sets, potentially leading to optimistic performance estimates due to information leakage.

While this evaluation protocol is consistent with the internal validation procedure typically employed within AutoML optimization loops, it does not fully guarantee independence between training and validation data. Therefore, the reported performance should be interpreted as an estimate of model capability under these conditions rather than as a fully unbiased generalization measure.

More rigorous evaluation protocols, such as temporally blocked splits or subject-independent validation strategies, will be considered in future work to provide a more robust assessment of generalization performance.

The resulting performance metrics are then used within the AutoML loop to update the surrogate model estimating the relationship between hyperparameter configurations and classification error.

Based on this surrogate model and the selected acquisition function, Auto-Weka identifies new promising configurations that balance exploration of previously untested regions of the search space and exploitation of configurations with low estimated error. These newly selected configurations are iteratively trained and evaluated, and the optimization loop is repeated until the predefined computational budget is exhausted. The budget can be specified either as a maximum execution time or as a memory constraint.

Once the optimization process terminates, Auto-Weka outputs the best configuration found over all evaluated candidates, corresponding to the model and hyperparameter set that achieved the lowest validation error within the given budget.

4.2. Hardware-Aware C++ Implementation

A significant challenge in the proposed pipeline arises from the architectural discrepancy between the exploration and implementation environments. Auto-Weka, being natively built upon the WEKA ecosystem, is a purely Java-based tool. Upon identifying the optimal classifier and hyperparameter configuration, it provides a model representation compatible with the Java Virtual Machine (JVM) and code snippets designed for the WEKA API. But the subsequent High-Level Synthesis (HLS) stage requires input source code in C or C++. Since Auto-Weka lacks a native C/C++ export functionality, the second stage of our pipeline serves as a critical translation layer. This process involves more than a simple syntax conversion: it requires a manual re-implementation of the model's inference logic to ensure it adheres to the strict constraints of hardware synthesis.

Specifically, the mathematical structure of the identified model (e.g., the coefficients and decision boundaries of a Multinomial Logistic Regression) must be extracted from the Java output and ported into a Hardware-Aware C++ kernel. This kernel is purposely structured to avoid specific constructs such as dynamic memory allocation, recursion, and complex control flows, ensuring it can be successfully mapped onto the FPGA's reconfigurable fabric. This necessity for cross-language translation and hardware optimization is the primary driver for the human intervention required in this stage of the pipeline.

While there have been independent efforts to bridge the gap between Java-based environments and C++, such as the Weka-C++ project (<https://openhub.net/p/wekacpp>, accessed on 7 July 2026) these repositories often remain unofficial, incomplete, or unmaintained. Moreover, these implementations focus on general-purpose C++ and obviously do not account for the stringent syntactic and architectural constraints required by HLS tools, such as the aforementioned exclusion of dynamic memory and standard library dependencies as well. Consequently, relying on such libraries would introduce significant risks regarding functional correctness and hardware compatibility.

A potential alternative involves the use of the Predictive Model Markup Language (PMML), an XML-based industry standard designed for model interoperability [33]. In theory, a model exported from Weka in PMML format could be translated into C/C++ via specialized conversion tools. However, this approach introduces several layers of inefficiency when targeting FPGA-based acceleration. The necessity of a specialized translation layer is well-documented in the literature, as standard machine learning frameworks are not designed to meet the nanosecond-latency and resource constraints of FPGA hardware [34].

In particular, while PMML provides a standard for model exchange, its application in hardware synthesis is hindered by the complexity of the generated representations, which often lack the hardware-aware optimizations required for efficient edge deployment [35]. As in the previous case, the resulting source often relies on dynamic memory management or standard library dependencies that are incompatible with hardware logic synthesis. There could also be the issue of functional incompleteness [36], as many open-source converters do not fully support the specific mathematical operators required by some models identified by the AutoML stage.

In light of these limitations, the manual extraction and re-implementation strategy was prioritized over automated PMML conversion. This approach provided the granular control necessary to map hardware resources effectively, ensuring seamless compatibility with the AMD/Xilinx Vitis HLS workflow and the target architecture.

A secondary, more technical challenge lies in the extraction of the model's internal parameters. As stated before, Auto-Weka excels at identifying the optimal algorithm but its standard output is primarily intended for execution within the WEKA environment. Therefore it does not inherently provide a direct export of the internal learned parameters (such as the weights, biases, or coefficients) in a format ready for hardware synthesis.

One possibility to obtain these values is to perform a model inspection phase. Once the optimal configuration is identified, the trained model is programmatically analyzed within the WEKA environment—utilizing Java reflection or internal WEKA methods—to access its underlying data structures. The extracted parameters are then mapped into the C++ inference kernel previously developed, where they are hard-coded as static constant arrays.

An alternative way is to re-compute them using external libraries, that can be selected considering the model suggested by Auto-Weka. In our use case we selected sklearn, as discussed in Section 5, because it is one of the few libraries that provides an optimized implementation of Multinomial Logistic Regression and allows extracting the model's coefficients and biases in a simple way.

It should be noted that the proposed implementation relies on double-precision floating-point arithmetic, which is not typically optimized for FPGA-based inference. This choice was made to ensure numerical consistency with the software model generated during the AutoML stage, as tools such as scikit-learn operate natively in double precision and allow straightforward extraction of model parameters.

While this approach simplifies the integration between the AutoML output and the hardware implementation, it introduces a non-negligible overhead in terms of resource utilization and performance. It is well established in the literature that floating-point operators require substantially more hardware resources and exhibit higher latency due to their increased computational complexity and normalization logic. In particular, fixed-point arithmetic is known to provide faster execution and lower resource utilization, whereas floating-point implementations incur higher logic requirements and reduced performance [37,38]. In practice, this translates into typical overheads on the order of several times in both resource usage and latency, depending on the operator and precision.

In conclusion, the lack of a standardized, cross-platform code and parameter export in Auto-Weka explains why this stage remains a manual, human-in-the-loop and critical process. While the first stage is devoted to the model selection, the stage focuses on a model implementation in C++ that is FPGA-aware while preserving numerical consistency with the original software model. In this context, double-precision arithmetic is adopted to ensure compatibility with the parameters extracted from scikit-learn.

On the contrary, production-ready systems must investigate the adoption of fixed-point and quantized representations, which are expected to further improve performance and energy efficiency, at the cost of additional design complexity and potential impacts on model accuracy.

4.3. Hardware Deployment via Vitis High-Level Synthesis

In the last stage the focus shifts from model selection to efficient hardware deployment, using Field-Programmable Gate Arrays (FPGAs) as the target execution platform and Vitis High-Level Synthesis (HLS) as the enabling toolchain.

An FPGA is a reconfigurable digital device made up of three main components: a large array of Configurable Logic Blocks (CLBs), which contain Look-Up Tables (LUTs) for implementing combinational logic and Flip-Flops (FFs) for storing state; Dedicated arithmetic resources, such as Digital Signal Processing (DSP) slices, optimized for multiply–accumulate operations commonly used in signal processing and machine learning; and On-chip memory blocks (BRAMs), used to store intermediate data, parameters, or buffering structures with low access latency. These components are connected through a programmable routing fabric that allows designers to define custom data paths and exploit fine-grained parallelism. The logical architecture is shown in Figure 3.

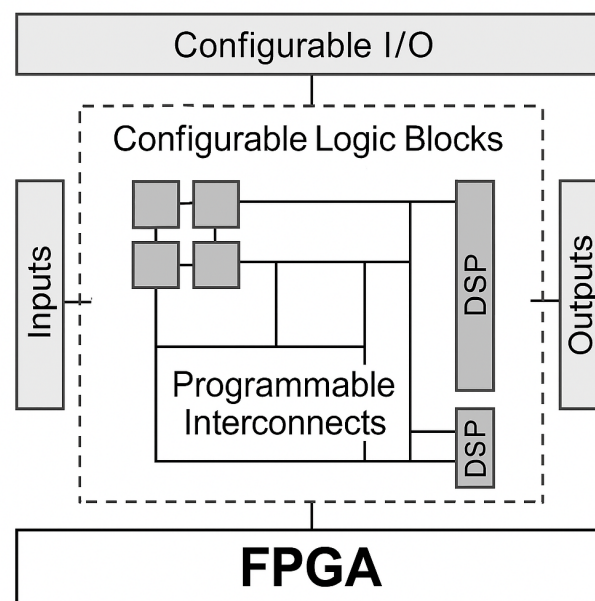


Figure 3. Generic logical architecture of a field-programmable gate array (FPGA). The device consists of many configurable logic blocks (e.g., the four dark squares) interconnected through a programmable routing fabric, complemented by dedicated DSP units and configurable I/O interfaces.

Unlike fixed-function processors, such as CPUs and GPUs, an FPGA allows the internal hardware structure to be tailored to a specific computational task. For machine learning inference, the key advantage of FPGAs lies not in high clock frequency, but in the ability to express spatial parallelism directly: independent operations can be mapped to distinct

hardware resources and executed concurrently, resulting in predictable timing behavior and high throughput at relatively low clock speeds.

Despite their advantages, FPGA-based design flows rely on a register-transfer level (RTL) abstraction to describe synchronous hardware, which represents the input expected by logic synthesis tools. RTL represents a fundamental level of abstraction in digital circuit design, in which the behavior of a synchronous system is described in terms of data transfers between registers and the logical or arithmetic operations applied to that data at successive clock cycles. Unlike lower levels of description, RTL does not focus on transistors or individual logic gates, but rather on the organization of data flow between memory elements and the transformations performed by the intervening combinational logic.

A typical RTL model is composed of three main elements. Registers are the memory elements responsible for holding data between one clock edge and the next, generally implemented using flip-flops. Combinational logic includes the set of logical and arithmetic functions that operate on data as it propagates from the outputs of source registers to the inputs of destination registers. Finally, the clock acts as a global synchronization signal, defining the discrete time instants at which the system state is updated.

RTL descriptions are typically expressed using hardware description languages (HDLs), such as Verilog and VHDL, which formally define the system's registers, data paths, and timing constraints. As such, RTL represents a point of convergence between functional design and physical circuit implementation: it is abstract enough to focus on system behavior, yet sufficiently detailed to be directly synthesized into hardware.

Listing 1 presents a basic RTL module written in Verilog that implements a synchronous counter. The *always @(posedge clk)* construct explicitly models clock-driven behavior, ensuring that state transitions occur only on the rising edge of the clock. Within this process, conditional control logic, such as reset handling, and arithmetic operations on registers collectively describe the behavior of both the sequential and combinational parts of the system. This example illustrates how an RTL description specifies data transfers and state updates in a timing-aware manner, rather than expressing computation as a sequential instruction stream as in software programs.

Listing 1: A 4-bit synchronous binary counter in Verilog.

```

module counter_4bit (
    input wire clk,
    input wire reset,
    output reg [3:0] count
);

always @(posedge clk) begin
    if (reset)
        //Reset counter to 0000
        count <= 4'b0000;
    else
        //Increment counter
        count <= count + 1'b1;
end
endmodule

```

Writing RTL code is inherently complex, as it requires a paradigm shift with respect to traditional software development and imposes strict physical, temporal, and methodological constraints on the designer. Moreover, debugging RTL designs is particularly challenging, since errors often emerge only after synthesis or during hardware execution and may be related to timing, concurrency, or resource conflicts rather than functional correctness.

For these reasons, direct RTL development represents a significant barrier in applied AI research, especially in edge scenarios where the primary focus lies on algorithm design and data processing rather than low-level hardware implementation. In the proposed pipeline, this complexity is addressed by relying on a combination of AMD/Xilinx Vitis High-Level Synthesis (HLS) and Vivado Design Suite, which jointly enable the transition from software-defined machine learning models to executable FPGA implementations.

Specifically, Vitis HLS operates at a higher level of abstraction, translating a hardware-aware C/C++ implementation of the selected model into a synthesizable RTL description. During this process, the tool automatically handles fundamental hardware design tasks such as operation scheduling, resource binding, and control logic generation, while allowing design constraints on clock frequency, latency, and throughput to guide the synthesis. Rather than replacing the traditional FPGA toolchain, Vitis HLS complements it.

The RTL and Intellectual Property (IP) cores generated by HLS serve as inputs to Vivado, which manages the downstream implementation phases, including logic synthesis, place-and-route, and timing closure on the target device. In this sense, the hardware deployment stage of the pipeline is not based on a single tool, but on the interplay between Vitis HLS and Vivado, where high-level architectural decisions are separated from device-specific physical realization. Although HLS tools are in principle vendor-agnostic, practical deployments typically rely on tightly integrated toolchains. The use of Vitis HLS and Vivado therefore reflects the dominance of AMD/Xilinx platforms in high-end FPGA deployments (<https://www.verifiedmarketresearch.com/blog/top-fpga-companies/>, accessed on 7 July 2026) rather than constraining the general applicability of the proposed approach.

The typical design flow for this stage can be summarized in the following steps [39]. It begins with the functional simulation step, whose purpose is to execute the implementation of the algorithm in order to validate its logical behavior prior to hardware synthesis. The C/C++ code is executed as a standard software program using a dedicated testbench, i.e., a verification environment responsible for providing input stimuli to the design under test and for checking the corresponding outputs against expected reference results. At this stage, no hardware-related aspects such as timing, parallelism, or resource utilization are considered. The goal is to validate the functional correctness of the algorithm and to detect subtle and hard-to-identify errors, such as those related to data dependencies, numerical precision, pipeline scheduling, or optimization side effects, which may not be evident through static code analysis alone.

Next, the validated high-level code is translated into an RTL representation, named *RTL creation* step, expressed in Verilog or VHDL. During this phase, the HLS tool analyzes the C/C++ code to derive a hardware architecture that implements the specified algorithm, explicitly defining registers, combinatorial logic, and control structures. The synthesis process is guided by a set of design constraints, including target clock frequency, latency, throughput, and area, which influence how operations are scheduled in time and mapped onto available hardware resources. Instruction-level operations described sequentially in C/C++ are reorganized into parallel hardware structures whenever data dependencies allow it. In this context, scheduling determines the clock cycle in which each operation is executed, binding assigns operations and data storage to specific hardware resources such as functional units and registers, and pipelining introduces temporal overlap between consecutive operations to increase throughput. The result of this phase is a synthesizable RTL description that captures both the functional behavior of the original algorithm and its timing-aware hardware realization, forming the basis for subsequent verification and physical implementation on the target FPGA.

The third step is the *C/RTL co-simulation*, whose purpose is to verify that the RTL representation generated by the HLS tool exhibits the same functional behavior as the

original C/C++ model. In this phase, the inputs produced by the C/C++ testbench are applied to both the software model and the synthesized RTL design, which is simulated using an HDL simulator. The corresponding outputs are then compared to ensure functional equivalence between the high-level description and its hardware implementation. This step is essential in edge AI applications, where even small numerical discrepancies can accumulate over time or affect classification reliability. Unlike the functional simulation step, this step takes into account hardware-specific aspects such as clocked execution and signal-level behavior, while still focusing on correctness rather than performance or resource optimization.

The fourth step is the *device-specific implementation*. The resulting IP core generated by the HLS tool is integrated into the conventional FPGA design flow. In this phase, the synthesizable RTL description produced by Vitis HLS is imported into a hardware design environment—in our case the Vivado Design Suite—where it is combined with the target device configuration and peripheral interfaces. Vivado is then used to perform logic synthesis, place-and-route, and timing closure, mapping the abstract RTL description onto the physical resources of the FPGA while ensuring that all timing constraints are satisfied. This step transforms the design from a technology-independent representation into a device-specific implementation, resulting in a finalized hardware configuration bitstream.

Once the implementation process is complete, the generated bitstream is deployed onto the target FPGA, enabling the execution of the accelerated hardware kernel in a real edge environment. This phase marks the transition from design-time validation to run-time operation, allowing the synthesized model to be evaluated in terms of latency, throughput, power consumption, and overall system behavior under realistic operating conditions.

Once the C++ model was validated functionally, Vitis HLS was used to synthesize the design for the target FPGA device. During synthesis, the tool provides detailed reports describing Estimated clock frequency, Latency in clock cycles for a single inference, initiation interval, which determines throughput, and the Utilization of hardware resources (LUTs, FFs, DSPs, BRAMs).

Using this feedback, the most computationally intensive parts of the model can be identified and appropriate countermeasures can be adopted. For example, in our use case, exponential and normalization operations associated with the softmax function were found to dominate latency. To mitigate this effect, loop pipelining directives were applied, allowing successive iterations of selected loops to overlap in time. Through iterative refinement guided by synthesis reports, a proper balance can be achieved between latency, throughput, and resource usage, resulting in hardware kernels well suited for edge deployment.

5. Experimental Results

5.1. AutoML Stage

To account for the stochastic nature of Bayesian optimization and to explore the impact of search budget, 10 independent Auto-Weka searches were performed, varying both the random seed and the time budget (30, 60, and 120 min).

This experimental setup was primarily intended to explore the impact of search duration and assess the consistency of the selected models across different configurations, rather than to provide a statistically exhaustive evaluation of convergence. While the number of runs is limited, the observed consistency in the selected model suggests robustness within the explored search space. Such constraints reflect practical design scenarios, where exploration time is limited by application requirements. On the contrary, stronger statistical

guarantees require more extensive experimental campaigns, including a higher number of runs and standardized time budgets.

As shown in Figure 2, we configured the other parameters as follows. The batchSize was fixed at 100 to align with standard predictive modeling practices, while the debug parameter was disabled to prevent the generation of redundant output data that would otherwise impede search efficiency. To further optimize search times, doNotCheckCapabilities was set to False; this ensures that Auto-Weka verifies classifier compatibility beforehand, avoiding the computational overhead of testing unsuitable models. Memory management was handled by allocating 2048 MB via the memLimit parameter, providing sufficient resources for each individual search process. The metric for model evaluation was defined as errorRate, as it directly aligns with the objective of maximizing predictive accuracy by identifying models with the lowest discrepancy. Finally, nBestConfigs was set to 1 to isolate only the highest-performing model, while the seed and timeLimit were the only variables adjusted across the ten distinct search iterations to ensure a comprehensive yet controlled exploration. The seed in fact affects the search initialization. The timeLimit was set to 30 min for 4 searches, 60 min for 4 searches and 120 min for 2 searches. This difference in the search time limit was necessary to understand how the Auto-Weka results change as the search time varies.

Upon completion of a search, Auto-Weka generates a text file containing the optimal model and its associated hyperparameters. This output also includes a set of instructions for replicating the model in Java using the Weka package (<https://weka.sourceforge.io/packageMetaData/>, accessed on 7 July 2026). However, this specific feature was not utilized in our workflow, as the subsequent integration of the model onto the FPGA required a C++ implementation rather than a Java-based environment, as said before.

The results across the 10 Auto-Weka iterations exhibited some variability, as the process did not converge on a single optimal model. Multinomial Logistic Regression (MLR) emerged as the most frequent recommendation, appearing in seven instances, followed by AdaBoost (twice) and Random Forest (once). Notably, the selection of MLR was consistently associated with the more extensive search durations; it was the result of both two-hour searches, three of the one-hour searches, and two of the thirty-minute iterations. Conversely, the AdaBoost and Random Forest algorithms were primarily identified during the shorter search windows and were characterized by inferior SMAC scores, suggesting that longer search times may lead to the identification of more competitive configurations within the explored search space. These results are shown in Table 1. It should be noted that the reported error rates were obtained using the AutoML validation protocol based on random sample-level data partitioning. Consequently, temporal correlations between training and validation samples may lead to optimistic performance estimates.

Table 1. The results of the 10 Auto-Weka searches.

Suggested Algorithm	Time Limit	Error Rate	SMAC Score
Random forest	30 min	0.023	0.108
Multinomial Logistic Regression	30 min	0.012	0.098
Multinomial Logistic Regression	30 min	0.016	0.099
AdaBoost	30 min	0.033	0.107
AdaBoost	60 min	0.0010	0.024
Multinomial Logistic Regression	60 min	0.0010	0.007
Multinomial Logistic Regression	60 min	0.0009	0.016
Multinomial Logistic Regression	60 min	0.0006	0.008
Multinomial Logistic Regression	120 min	0.0	0.006
Multinomial Logistic Regression	120 min	0.0	0.0059

Based on the analysis of these results, Multinomial Logistic Regression (MLR) emerged as the most robust and frequently selected candidate and was therefore adopted as the reference model for the subsequent implementation phase for the physiological dataset, supported by three primary factors. First, the algorithm demonstrated high consistency, being recommended in 70% of the search instances. Second, it was the model identified in the iteration yielding both the optimal SMAC score and the lowest estimated error rate. Finally, compared to the other suggested candidates, MLR offers a superior balance between robustness and computational efficiency; its lightweight architecture is particularly well-suited for the specific characteristics of the dataset and aligns with the resource constraints of the subsequent FPGA deployment phase. Consequently, this model was adopted for training and served as the foundation for the hardware porting process.

5.2. Hardware-Aware C++ Implementation Stage

Using sklearn's logistic regression function and the physiological dataset, three similar models were trained. The decision to train three models with different feature subsets was motivated by the need to analyze the trade-off between model complexity, predictive performance, and hardware efficiency.

The first MLR model was trained using the complete feature set of the physiological dataset to predict a subject's activity type during a specific observation. The input features included heart rate, breath frequency, respiration, ECG, R2R intervals, and three-axis acceleration (X, Y, and Z). Resulting coefficients and biases are shown in Table 2.

Table 2. The coefficients and the biases found for the MLR model with all the 8 features. In each row there are coefficients for each class, in the columns there are coefficients for each feature and in the last column there are biases for each class.

HR	BF	RES	ECG	R2R	acc X	acc Y	acc Z	Biases
−0.1307	0.2265	−0.0559	0.22214	2.0489	1.1138	0.2243	4.90×10^{-5}	−277.77
−0.4122	−0.3357	−0.3839	0.1693	1.2017	−0.740	−0.0524	−0.0002	164.22
0.5987	−0.2875	0.5608	0.1859	0.1236	−0.0744	−0.0295	7.76×10^{-5}	−15.489
−0.2175	0.3562	−0.0254	−0.292	−4.805	−0.0121	−0.079	2.61×10^{-5}	32.194
0.1618	0.0406	−0.0956	−0.284	1.430	−0.287	−0.0624	$−8.27 \times 10^{-5}$	96.846

The second model was trained on only five features from the dataset: heart rate, breath frequency, respiration, ECG and R2R. Excluding the acceleration features was primarily to create a lighter model, as the discarded features did not significantly affect the predictive model's performance. Resulting coefficients and biases are shown in Table 3.

Table 3. The coefficients and the biases found for the model with 5 features. In each row there are coefficients for each class, in the columns there are coefficients for each feature and in the last column there are biases for each class.

HR	BF	RES	ECG	R2R	Biases
0.220051	2.04891	1.11017	0.223619	0.000117199	−276.6576
0.174333	1.23153	−0.734954	−0.0521767	−0.00011863	163.0590
0.181139	0.130759	−0.0764211	−0.0295454	0.00014559	−15.4323
−0.291914	−4.81771	−0.0132671	−0.0797118	9.41345×10^{-5}	32.5635
−0.28361	1.40651	−0.285523	−0.062185	$−1.4214 \times 10^{-5}$	96.4673

At last, the third model was trained on only three features of the dataset: heart rate, breath frequency and respiration. The purpose of this model was to evaluate the impact of a highly lightweight configuration on FPGA performance. As expected, this model exhibits

lower predictive performance compared to the more complex configurations. At the end of training, the coefficients and biases listed in Table 4 were found.

Table 4. The coefficients and the biases found for the model with 3 features. In each row there are coefficients for each class, in the columns there are coefficients for each feature and in the last column there are biases for each class.

HR	BF	RES	Biases
0.0509846	0.391408	0.0798768	−89.31607443
0.212564	−0.169964	−0.00882437	19.25098301
0.0878693	−0.139796	−0.0182498	23.97523702
−0.174342	−0.0178114	−0.0283784	21.33804296
−0.177076	−0.0638371	−0.0244242	24.75181145

Once the coefficients and biases for the three models trained in Python 3.10 were obtained, the three predictive models were developed in C++. The implementation was constrained to use only standard C++ constructs to ensure compatibility with the Vitis HLS toolchain and enable efficient hardware synthesis. The code for the three models is very similar. An example is provided in Listing 2.

All three prediction models were trained on 80% of the dataset and evaluated on the remaining 20% using standard sklearn functions such as fit() and score(). The validation results are shown in Table 5.

Listing 2: The FPGA-compliant MLR implementation with 5 features.

```

1  #include <array>
2  #include <iostream>
3  #include <cmath>
4  #include <cstdint>
5  #include <algorithm>
6  #include <chrono>
7
8  int predict_model(const std::array<double,5>& x) {
9      double coeffs[5][5] = {
10         {2.20051328e-01,  2.04890955e+00,  1.11016533e+00,  ...},
11         ...
12         {−2.83609800e-01,  1.40650893e+00,  −2.85522588e-01,  ...}
13     };
14     double biases[5] = {
15         −276.65764851, 163.05908344, −15.43233559,
16         32.56356667,  96.46733399
17     };
18
19     std::array<double,5> y;
20     for (std::size_t i = 0; i < 5; ++i) {
21         y[i] = coeffs[i][0]*x[0] + coeffs[i][1]*x[1] +
22             coeffs[i][2]*x[2] + coeffs[i][3]*x[3] +
23             coeffs[i][4]*x[4] + biases[i];
24     }
25
26     double Z = 0.0;
27     for (std::size_t i = 0; i < 5; ++i) {
28         y[i] = std::exp(y[i]);
29         Z += y[i];
30     }
31
32     for (std::size_t i = 0; i < 5; ++i) {
33         y[i] /= Z;
34     }
35     auto it = std::max_element(y.begin(), y.end());

```

```

36     auto idx = std::distance(y.begin(), it);
37
38     return idx+1;
39 }

```

Then, the average prediction speed of each model has been measured. The tests were performed to understand the performance of the three models executed sequentially within the CPU, as these speed tests are necessary for comparison with their optimized versions after the porting step. The evaluation was performed over the entire dataset (18,422 samples), measuring the prediction times for each observation's category and finally averaging all the estimated times. The test results are shown in Table 5. The reported classification scores were obtained under the same validation protocol adopted within the AutoML process. As discussed before, random sample-level splits may introduce temporal correlations between training and validation data; therefore, the reported scores should be interpreted accordingly.

Table 5. The validation scores of the three Multinomial Logistic Regression models and the execution times, in nanoseconds, of the C++ implementation. Latency measurements were repeated multiple times and averaged to reduce measurement noise.

MLR	Score	Execution Time
8 features	0.9901	433
5 features	0.9902	415
3 features	0.8217	298

5.3. Hardware Deployment Stage

5.3.1. Functional Simulation

To identify architectural bottlenecks and ensure functional correctness, the implementation underwent a rigorous validation phase using the Vitis Code Analyzer and a custom-developed C++ testbench. The testbench is responsible for verifying that the model's outputs remain consistent across all synthesis steps.

The testbench was executed using a dataset of 1000 input samples, previously evaluated by the CPU-based reference models. This sample size was selected to provide statistical confidence in the results while maintaining manageable synthesis and simulation times, as larger datasets significantly latency-penalize the iterative hardware development flow.

The initial analysis identified two critical sections within the C++ source code that inhibited full hardware optimization. The first issue involved the multiplication of input features by the coefficient matrix (lines 20–24 of Listing 2). The Vitis Code Analyzer flagged this region due to resource contention: the variables x and $coeffs$ were targeted by multiple simultaneous access requests that the underlying hardware ports could not satisfy within a single clock cycle. To resolve this and enable parallel execution, the loop was manually unrolled as follows:

```

y[0] = coeffs[0][0]*x[0] + ... + coeffs[0][4]*x[4] + biases[0];
...
y[4] = coeffs[4][0]*x[0] + ... + coeffs[4][4]*x[4] + biases[4];

```

By explicitly defining each operation, the HLS compiler can better map these accesses to independent memory banks or registers, effectively accelerating the inference kernel.

A second bottleneck was identified in the probability calculation phase (lines 27–30 of Listing 2). A cyclic data dependency (Read-After-Write hazard) occurred because the variable y was being updated and accumulated within the same loop structure. This dependency prevented the compiler from overlapping iterations. To eliminate this hazard,

the exponential calculation was decoupled from the accumulation of the normalization factor Z , allowing for a more efficient hardware mapping, as follows:

```
for (std::size_t i = 0; i < 5; ++i) {
    y[i] = std::exp(y[i]);
}
double Z = y[0] + y[1] + y[2] + y[3] + y[4];
```

Once these structural issues were resolved, the focus shifted to optimizing the Initiation Interval (II). In HLS, the II represents the minimum number of clock cycles between successive executions of a process. Since the overall system throughput is dictated by the slowest component in the dataflow, optimization efforts were concentrated on the processes with the highest IIs.

The Code Analyzer decomposed the sequential C++ description into seven concurrent processes, as illustrated in the dataflow graph in Figure 4a. Initial estimates showed that Process 2 (exponential calculation) and Process 4 (normalization/division) were the primary bottlenecks, with IIs of 297 and 302 clock cycles (CC), respectively.

To mitigate these latencies, the `#pragma HLS PIPELINE II=1` directive was applied to the internal loops of these processes. By setting the II to 1, the hardware is instructed to begin a new iteration every clock cycle, maximizing throughput. Following this optimization, the II for Process 4 dropped from 302 to 67 CC, while Process 2 saw a reduction from 297 to 66 CC. As shown in Figure 4b, the final optimized architecture exhibits balanced initiation intervals across all processes, with the worst-case II reduced to 67 CC, representing a significant increase in overall system throughput.

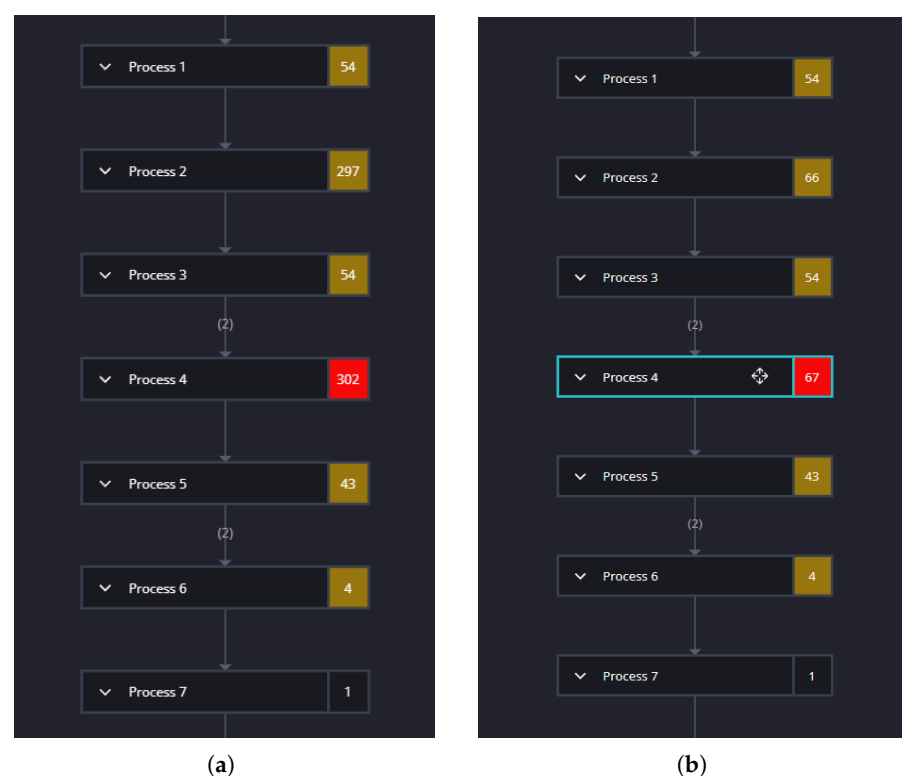


Figure 4. (a) A graph of the processes with their IIs generated by Vitis HLS. (b) A graph of the processes with their IIs generated by Vitis HLS after the optimization.

5.3.2. RTL Creation

This step translates the optimized high-level C++ description obtained in the previous step into a Register-Transfer Level (RTL) representation suitable for hardware implemen-

tation. In our test the 43 lines of C++ code have been translated to 5805 lines of code in VHDL.

The RTL generation process is guided by the constraints of the target FPGA device, the specified clock frequency, and the selected output flow. In this work, the flow target was set to Vivado, as it provides the full toolchain required for downstream implementation steps, including logic synthesis, placement, routing, and timing analysis.

Upon completion of high-level synthesis, Vitis HLS produces a comprehensive set of analysis reports accessible through the Flow Navigator of the Vitis HLS IDE. These reports are essential for assessing both performance and resource efficiency of the generated RTL design. In particular, the Module Hierarchy view illustrates the structural organization of RTL blocks, highlighting latency contributions and potential timing violations. The Performance Profile provides detailed metrics on loop pipelining behavior and initiation intervals, while the Resource Profile maps hardware resources to specific control states. Additionally, the Properties View offers fine-grained metadata for individual operations selected within the scheduling environment.

A key component of the analysis is the Schedule Viewer, which presents a cycle-accurate representation of the synthesized RTL by mapping each operation and control step to its corresponding execution cycle. This visualization is instrumental in identifying loop-carried dependencies that limit parallelism, as well as datapath bottlenecks that impact overall throughput. As shown in Figure 5, the vertical axis represents the RTL hierarchy in topological order, ensuring that data dependencies propagate from preceding to subsequent operations. Operations are depicted as gray boxes whose horizontal extent corresponds to their latency relative to the clock period. Multi-cycle operations are indicated by a central horizontal line. Data dependencies are rendered as solid blue connections; selecting an operator highlights its predecessors and successors, enabling detailed inspection of the critical path and facilitating informed optimization decisions.

During the initial synthesis iterations, timing violations were observed due to combinational paths that failed to meet the default clock constraint. To achieve timing closure, the target clock period was adjusted according to model complexity: 3 ns for the 3-feature model, 4 ns for the 5-feature model, and 5 ns for the 8-feature model. These adjustments successfully eliminated timing violations while preserving optimal throughput.

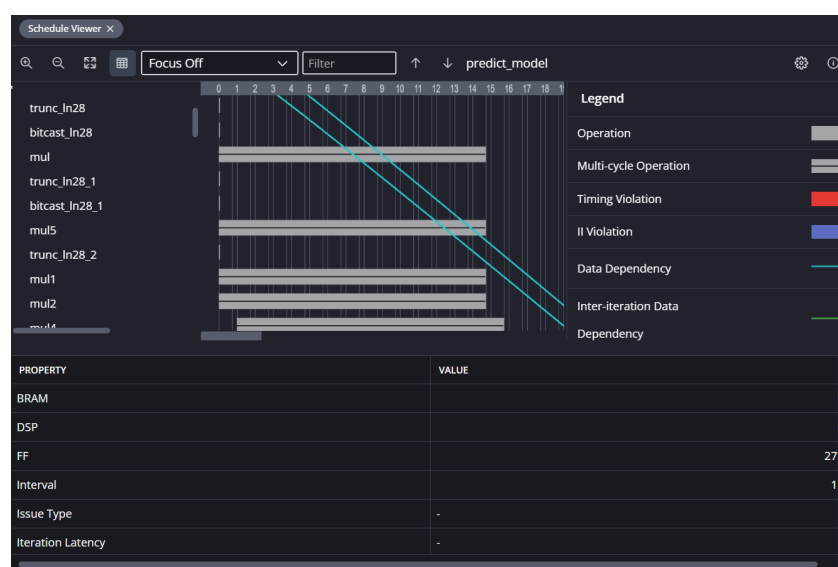


Figure 5. Vitis HLS Schedule Viewer for the 3-feature MLR model.

5.3.3. C/RTL Co-Simulation

The functional equivalence between the high-level C++ implementation and the synthesized RTL design was validated through C/RTL co-simulation, using the previously described testbench. This verification step is available within the Flow Navigator of the Vitis HLS IDE, where the Run Co-Simulation command enables comprehensive assessment of the RTL synthesis results.

The co-simulation process is structured into three sequential phases. First, a C-level simulation is executed, during which the testbench applies predefined stimuli to the top-level function, referred to as the Design Under Test (DUT). The input values provided to the DUT are recorded and stored as input vectors. Second, these input vectors are reused to drive an RTL simulation executed within the Vivado simulator, using the RTL code generated by Vitis HLS. The resulting outputs of the RTL simulation are captured and stored as output vectors. Finally, the output vectors are returned to the main() function of the C testbench, which verifies their correctness by comparing them against the expected reference results.

Upon completion of the co-simulation, Vitis HLS automatically generates a comprehensive set of reports, including co-simulation summaries, a Timeline Trace, and a Function Call Graph. It is worth noting that the performance metrics obtained from C/RTL co-simulation may differ from those reported after HLS synthesis, as the latter are derived from theoretical best- and worst-case path evaluations. In contrast, co-simulation results reflect the actual latency and initiation interval (II) observed for the specific dataset used during simulation, thereby providing a more realistic characterization of runtime behavior.

As shown in Figure 6, the Timeline Trace Viewer provides a detailed temporal profile of the execution of functions within the implemented design. This visualization enables analysis of function concurrency, execution overlap, and runtime distribution, and is instrumental in validating scheduling decisions and confirming that the synthesized RTL meets the intended performance objectives.

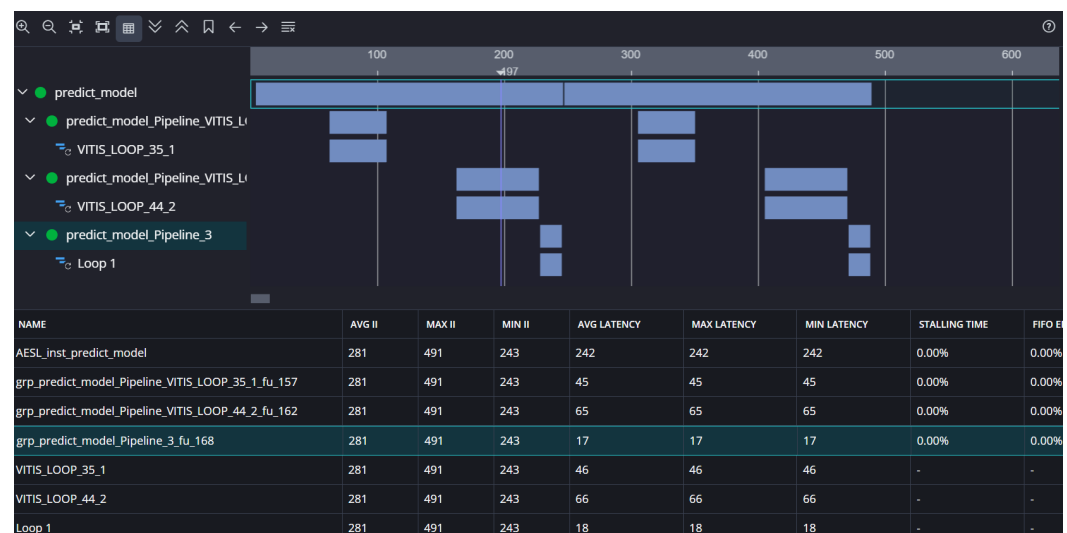


Figure 6. Timeline trace of the 3-feature MLR model on 1000 instances.

5.3.4. Device-Specific Implementation

The estimations provided by Vitis HLS regarding the generated RTL design are inherently limited, as they are based on analytical projections rather than on a full physical implementation. While the tool can estimate resource utilization and timing characteristics of the synthesized design, these values represent only predicted outcomes. To obtain an accurate assessment of the RTL design, the generated code must be processed

through Vivado logic synthesis followed by place-and-route. This step enables evaluation of the actual timing closure and hardware resource utilization, as reported by the Vivado implementation tools, thereby providing a realistic characterization of the final hardware design.

In this work, model acceleration is performed on an AMD/Xilinx Artix UltraScale+ XCAU20P-2SFVB784I FPGA (<https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/artix-ultrascale-plus.html>, accessed on 7 July 2026), a medium-level device targeting edge and embedded scenarios characterized by strict constraints on latency, energy consumption, and computational efficiency.

The Implementation Report contains the results of the Synthesis and Place-and-Route considering that FPGA. The most interesting values are shown in Table 6 and in Figure 7.

Table 6. Table of results after the simulation on Vivado.

	3-Feature Model	5-Feature Model	8-Feature Model
LUT usage	9492	10,134	11,420
LUT percentage	10.44%	10.46%	10.48%
FF usage	15,923	16,737	11,595
FF percentage	7.30%	7.67%	5.32%
DSP usage	83	111	121
DSP percentage	9.22%	12.33%	13.44%
BRAM usage	2	2	2
BRAM percentage	1%	1%	1%

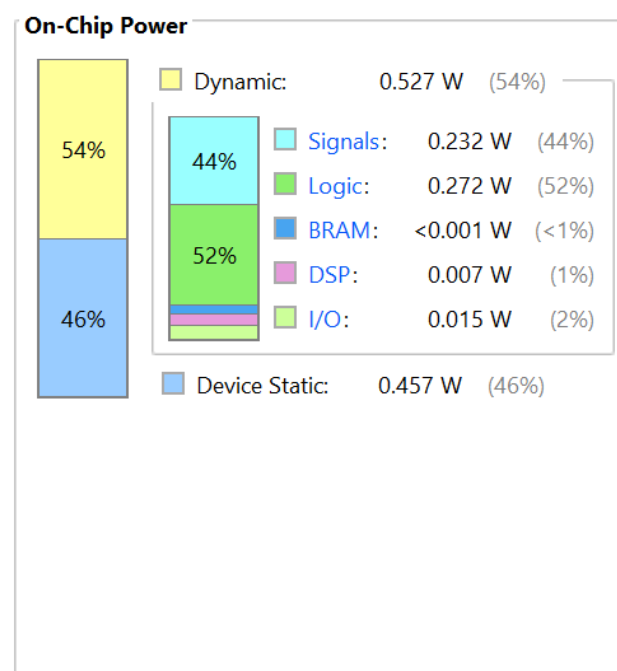


Figure 7. The power consumption diagram that occurs while using the FPGA board while running the MLR model with 3 features. Power consumption is divided into static and dynamic. Static power consumption occurs from the moment the board is powered on, while dynamic power consumption occurs only while the model is running.

5.4. Execution

The execution times and power consumption values have been collected for the three Multinomial Logistic Regression (MLR) models considering the FPGA board, a Raspberry Pi 5 and a medium-level laptop equipped with an AMD Ryzen 7 4800H.

The CPU requires 298 ± 5.1 ns for the 3-feature model, 415 ± 7.8 ns for the 5-feature model, and 433 ± 11.2 ns for the 8-feature model. The FPGA board requires 272 ± 4.5 ns for the 3-feature model, 377 ± 8.9 ns for the 5-feature model, and 394 ± 14.6 ns for the 8-feature model, as shown in Table 7. These values increase with model dimensionality, reflecting the additional arithmetic operations required to process a larger number of features. A comparison with the corresponding CPU-based implementation shows that FPGA execution achieves a consistent performance improvement of approximately 10% across all models. In absolute terms, the FPGA reduces execution latency by 26 ns for the 3-feature model, 38 ns for the 5-feature model, and 39 ns for the 8-feature model. These improvements remain larger than the measured standard deviations on both platforms, suggesting that the observed gains are not attributable to timing fluctuations alone.

In contrast, when compared to a resource-constrained platform such as the Raspberry Pi 5, the FPGA provides significantly larger latency reductions, ranging from approximately 37% to 52% across the considered models. It should be noted that all measurements were obtained over 10 runs for each configuration (hardware platform and MLR model). The reported FPGA latencies do not include host $\leftrightarrow$ FPGA data transfer overheads. For both the CPU and Raspberry Pi 5 implementations, the code was compiled using *gcc* -O3 optimizations. Although the absolute latency improvements are relatively modest in the high-performance CPU comparison, the relative speedup remains stable across configurations, indicating predictable scaling behavior of the proposed approach.

Focusing on the CPU, the modest magnitude of the speedup can be attributed to the low computational complexity of the MLR models, which are already efficiently executed on the CPU. However, this simplicity also translates into limited hardware utilization on the FPGA. As reported in Table 6, the 3-feature MLR implementation uses only 10.44% of the available LUTs, 7.3% of Flip-Flops, 9.22% of DSP blocks, and approximately 1% of BRAM resources. Similar utilization levels were observed for the other models, with resource usage remaining well below the FPGA's capacity.

From a quantitative perspective, these figures imply that nearly 90% of the LUTs and over 90% of the DSP resources remain unused when deploying a single instance of the model. This underutilization highlights the potential for spatial parallelism, because it suggests that multiple independent instances could be instantiated—up to 9 in our case—enabling effective exploitation of spatial parallelism. This is one of the key objectives of the considered use case, i.e., the possibility to manage multiple data flow streamed by different persons using a single edge device, as discussed in the next Section.

To quantify the benefits of parallel execution, aggregate throughput (inferences per second) is computed for both CPU and FPGA configurations. For the 8-feature model, the CPU achieves a throughput of approximately 2.31 M inf/s, while the FPGA reaches 17.8 M inf/s in the multi-instance configuration, corresponding to a throughput speed-up of about 7.7 $\times$.

The use of FPGA acceleration not only provides a measurable improvement in execution performance, but also enables substantially higher energy efficiency compared to the CPU-based implementations on both the laptop and the Raspberry Pi 5.

As regards the laptop, the AMD Ryzen 7 4800H presents a peak power consumption of approximately 60–65 W under full load. However, this value corresponds to full multi-core utilization and does not reflect the actual power consumption during single-thread inference. For the considered workload CPU power was measured in the range 15–21 W under single-thread workload conditions using system-level monitoring tools. In contrast, the Raspberry Pi 5 platform exhibits a significantly lower power consumption, ranging between 2.134 W and 2.622 W under the tested conditions.

As reported in Table 7, the FPGA implementation operates within a comparable power range for single-instance execution (approximately 1–1.6 W), while multi-instance configurations increase consumption up to about 11 W. Despite this increase, the FPGA achieves substantially higher throughput, resulting in superior energy efficiency compared to both desktop CPU and embedded platforms.

Table 7. Experimental results for single-instance and multi-instance FPGA execution compared with the use of a medium-level laptop and a Raspberry Pi 5. To better reflect the benefits of parallel execution, aggregate throughput (million inferences per second) is reported for multi-instance configurations with respect to the CPU execution.

	3 f. Model	5 f. Model	8 f. Model
Exec. time (CPU)	298 ns	415 ns	433 ns
Power (CPU)	15.6 W	17.1 W	20.8 W
Energy (CPU)	4.65 μ J/inf	7.10 μ J/inf	9.01 μ J/inf
Exec. time (RPI5)	563 ns	601 ns	703 ns
Power (RPI5)	2.134 W	2.408 W	2.622 W
Energy (RPI5)	1.20 μ J/inf	1.45 μ J/inf	1.84 μ J/inf
Exec. time (1 FPGA)	272 ns	377 ns	394 ns
Power (1 FPGA)	0.963 W	1.236 W	1.557 W
Energy (1 FPGA)	0.26 μ J/inf	0.47 μ J/inf	0.61 μ J/inf
FPGA instances (N)	9	8	7
Throughput (N FPGA)	33.1 M inf/s	21.2 M inf/s	17.8 M inf/s
Throughput (CPU)	3.35 M inf/s	2.41 M inf/s	2.31 M inf/s
Power (N FPGA)	5.01 W	6.69 W	8.16 W

It is important to distinguish between the methodologies used for timing and power evaluation of the FPGA board. Execution latencies were obtained through direct measurement, while power consumption was estimated using the Vivado post-implementation power analysis tool. Since post-implementation estimates were available only for the single-instance configurations, multi-instance power consumption was estimated by separating static and dynamic power contributions and assuming that only the dynamic component scales with the number of instantiated kernels. Additional power contributions arise from increased routing complexity, clock distribution loading, and interconnect activity, all of which become more significant as the number of parallel instances grows. Consequently, multi-instance power consumption is estimated by assuming that only the dynamic power component scales with the number of instantiated kernels, while the static power component remains unchanged. Using the static power estimate of 0.457 W reported in Figure 7, the total power consumption of an N-instance configuration is computed as $P_N = P_{static} + N \cdot P_{dynamic}$.

To facilitate a fair comparison across platforms, Table 7 also reports the energy per inference, computed as the product of execution latency and power consumption. This metric normalizes energy consumption with respect to the amount of computation performed and provides a more meaningful assessment of energy efficiency than power values alone.

The observed energy efficiency gains can be attributed to several architectural characteristics of FPGA devices. First, the algorithm is mapped directly onto dedicated hardware structures, eliminating instruction fetch and control overhead typical of general-purpose processors. Second, FPGA designs typically operate at lower clock frequencies than CPUs, reducing dynamic power consumption. Third, only the hardware resources actively involved in the computation are utilized, while unused components remain inactive. Finally, the FPGA implementation minimizes external memory access by leveraging on-chip resources and streaming dataflows, further improving efficiency.

Overall, these factors contribute to the superior performance-per-watt characteristics of the FPGA-based solution.

6. Discussion

The experimental results provide several insights into the behavior and limitations of the proposed AutoML-to-FPGA workflow under realistic edge AI constraints.

First, the observed latency improvements on FPGA are relatively modest when compared to a high-performance CPU (approximately 10%). This result can be attributed to the low computational complexity of the selected MLR model and the use of double-precision floating-point arithmetic, which introduces additional overhead in FPGA implementations. In this configuration, the FPGA does not fully exploit its parallel processing capabilities at the single-instance level.

A different trend emerges when considering parallel execution. The ability to replicate multiple inference kernels allows the FPGA to effectively leverage spatial parallelism, leading to a significant increase in aggregate throughput. This behavior highlights an important characteristic of FPGA-based acceleration: performance gains are primarily achieved through concurrency rather than single-instance latency reduction. As a result, the FPGA architecture is particularly well-suited for workloads involving multiple independent data streams, where throughput scalability is more critical than per-sample latency.

From an energy perspective, the evaluation based on energy per inference provides a more meaningful comparison across platforms. While raw power values differ significantly between CPU, embedded platforms like Raspberry Pi 5, and FPGA implementations, normalizing power consumption by execution time shows that the FPGA achieves substantially higher energy efficiency, particularly under parallel workloads. This advantage is especially relevant in edge scenarios, where continuous operation under constrained power budgets is required.

From a design standpoint, the results highlight the continued importance of hardware awareness in High-Level Synthesis (HLS)-based workflows. Although modern HLS tools abstract many low-level hardware details, achieving efficient implementations still requires careful consideration of control flow regularity, loop structure, and data movement. Minimizing dynamic control constructs and favoring deterministic execution patterns is essential to enable effective scheduling and pipelining. These constraints directly influence the types of models that can be efficiently mapped onto FPGA architectures.

In this context, the reliance on a simple model such as MLR should be interpreted as a deliberate design choice. The use of a well-structured and computationally lightweight model enables a clearer evaluation of system-level aspects of the proposed workflow, avoiding confounding effects introduced by model-specific optimizations. At the same time, this choice limits the extent to which absolute performance gains can be demonstrated.

More complex model families, such as decision trees, ensemble methods (e.g., Random Forests and AdaBoost), and neural networks, have already been successfully deployed on FPGA platforms using dedicated frameworks. Tools such as hls4ml (<https://fastmachinelearning.org/hls4ml/>, accessed on 7 July 2026) enable the translation of neural networks into hardware implementations, while libraries like Conifer (<https://github.com/thesps/conifer>, accessed on 7 July 2026) support low-latency deployment of decision forest models. However, these approaches are typically tailored to specific model classes and rely on dedicated conversion pipelines.

In contrast, general-purpose AutoML frameworks such as Auto-Weka do not provide a standardized export interface or hardware-ready representation for the wide variety of models they generate. As a consequence, extending an AutoML-driven workflow to FPGA deployment requires model-specific adaptation and additional design effort. While the

proposed pipeline is not inherently limited to linear models, supporting more complex algorithms would introduce additional challenges related to irregular control flow, memory access patterns, and resource management.

It is also important to distinguish the proposed approach from hardware-aware AutoML and Neural Architecture Search (NAS) frameworks such as APNAS and CoDAC presented in the Related Works Section. These methods focus on incorporating hardware constraints directly into the model design and training process, typically targeting neural network architectures and optimizing them for specific hardware platforms.

In contrast, the methodology proposed in this work addresses a complementary challenge, namely the integration of general-purpose AutoML frameworks with FPGA-based deployment flows. Rather than modifying the learning process itself, the proposed pipeline focuses on bridging the gap between software-level model generation and hardware-level implementation using existing toolchains.

As a result, the two approaches operate at different levels of the design stack and are not directly comparable in terms of experimental metrics. While hardware-aware AutoML frameworks optimize model architectures for given hardware constraints, the proposed pipeline emphasizes practical deployment, interoperability, and system-level integration across heterogeneous environments. This distinction clarifies that the contribution of this work lies in system-level integration rather than in hardware-aware model design, complementing existing approaches rather than directly competing with them.

Overall, the current implementation should be interpreted as a proof of concept that validates the integration of AutoML-based model selection with FPGA-based deployment using a representative model. While the methodology is designed to be extensible, further validation across different model families, datasets, and hardware platforms is required to fully assess its general applicability.

From a system-level perspective, the results also reveal opportunities for improved resource utilization. The observed underutilization of FPGA resources in the selected configuration suggests that future designs could emphasize throughput-oriented architectures, where multiple model instances or heterogeneous accelerators coexist on the same device.

Finally, although the proposed workflow significantly lowers the barrier to FPGA-based deployment by combining AutoML with HLS, it does not eliminate the need for hardware-aware design expertise. Effective exploitation of FPGA capabilities still requires informed design decisions, particularly when interpreting synthesis reports, resolving timing constraints, and optimizing resource utilization.

Taken together, these observations indicate that the proposed pipeline provides a practical and flexible foundation for edge-oriented machine learning deployment, while also highlighting key challenges that must be addressed to achieve fully automated and scalable ML-to-hardware integration.

7. Conclusions and Future Directions

This work addressed the growing demand for edge-centric artificial intelligence systems capable of operating under strict constraints on latency, energy consumption, and real-time responsiveness.

To this end, a pipeline was proposed to integrate Automated Machine Learning (AutoML) with High-Level Synthesis (HLS) and FPGA-based acceleration, with the goal of bridging the gap between model selection and hardware deployment in edge AI scenarios.

The experimental results show that AutoML can effectively identify models that balance predictive performance and computational efficiency. In the considered application, Auto-Weka consistently selected a Multinomial Logistic Regression (MLR) model, demon-

strating that relatively simple models can satisfy real-time constraints while remaining compatible with efficient hardware implementation.

From a system perspective, the FPGA implementation exhibited modest latency improvements at the single-instance level ($\approx 10\%$ compared to CPU execution), but significantly larger gains when compared to resource-constrained edge platforms such as the Raspberry Pi 5. More importantly, substantial performance improvements emerge when exploiting parallelism. By replicating multiple inference kernels, the system achieved more than a $7\times$ increase in throughput, confirming that FPGA acceleration is particularly effective for multi-stream, throughput-oriented workloads typical of edge computing scenarios.

In addition, when evaluated in terms of energy per inference, the FPGA implementation achieves more than an order of magnitude improvement in energy efficiency compared to CPU-based solutions, highlighting its suitability for continuous, low-power operation.

At the same time, the study reveals key limitations of current AutoML-to-hardware workflows. In particular, the lack of standardized model representations across heterogeneous toolchains requires a manual, hardware-aware re-implementation step, making the overall process inherently hybrid and dependent on domain-specific expertise.

It should also be noted that the experimental validation is limited to a single application scenario, model family, and FPGA device. Therefore, the results should be interpreted as a proof of concept rather than a comprehensive demonstration of general applicability.

Future work will focus on several complementary directions. First, extending the pipeline to support more complex model families (e.g., ensemble methods and neural networks) will be essential to assess its scalability across a broader range of applications. Second, integrating hardware-aware cost functions within the AutoML process could enable automated co-design, directly optimizing models for FPGA-level metrics such as latency, resource utilization, and energy efficiency. Finally, exploring standardized intermediate representations and cross-platform deployment strategies will be key to improving portability and reducing the need for manual intervention.

Author Contributions: Conceptualization, Y.S., G.D. and D.D.; methodology, Y.S.; software, Y.S.; validation, Y.S., G.D. and D.D.; data curation, Y.S.; writing—original draft preparation, Y.S.; writing—review and editing, G.D. and D.D.; supervision, G.D. and D.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Dataset available on request from the authors.

Acknowledgments: The authors would like to thank Francesco Ancarani, Daniele Grosso and Luca Noli from Insyde for their valuable technical support and expertise in FPGA architectures and hardware acceleration, which significantly contributed to the development and validation of this work.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Gill, S.S.; Golec, M.; Hu, J.; Xu, M.; Du, J.; Wu, H.; Walia, G.K.; Murugesan, S.S.; Ali, B.; Kumar, M.; et al. Edge AI: A taxonomy, systematic review and future directions. *Clust. Comput.* **2025**, *28*, 18. [[CrossRef](#)]
2. Ferrari, M.; D’Agostino, D.; Aguzzi, J.; Marini, S. Underwater Mediterranean image analysis based on the compute continuum paradigm. *Future Gener. Comput. Syst.* **2025**, *162*, 107481. [[CrossRef](#)]
3. Ficili, I.; Giacobbe, M.; Tricomi, G.; Puliafito, A. From Sensors to Data Intelligence: Leveraging IoT, Cloud, and Edge Computing with AI. *Sensors* **2025**, *25*, 1763. [[CrossRef](#)] [[PubMed](#)]

4. Shankar, V. Edge AI: A comprehensive survey of technologies, applications, and challenges. In *Proceedings of the 2024 1st International Conference on Advanced Computing and Emerging Technologies (ACET)*; IEEE: New York, NY, USA, 2024; pp. 1–6.
5. Cordova-Cardenas, R.; Amor, D.; Gutiérrez, Á. Edge AI in Practice: A Survey and Deployment Framework for Neural Networks on Embedded Systems. *Electronics* **2025**, *14*, 4877. [\[CrossRef\]](#)
6. Alcobaça, E.; de Carvalho, A.C. A literature review on automated machine learning. *Artif. Intell. Rev.* **2026**, *59*, 1–39.
7. Paleyes, A.; Urma, R.G.; Lawrence, N.D. Challenges in deploying machine learning: A survey of case studies. *ACM Comput. Surv.* **2022**, *55*, 5. [\[CrossRef\]](#)
8. Mao, Y.; Yu, X.; Huang, K.; Zhang, Y.J.A.; Zhang, J. Green edge AI: A contemporary survey. *Proc. IEEE* **2024**, *112*, 880–911. [\[CrossRef\]](#)
9. Bricola, A.; Noceti, N.; D’Agostino, D. Performance analysis of Convolutional Neural Networks on edge devices for Computer Vision tasks. *Array* **2026**, *29*, 100692. [\[CrossRef\]](#)
10. González, M.L.; Ruiz, J.; Andrés, L.; Lozada, R.; Skibinsky, E.S.; Fernández, J.; Sedano, J.; García-Vico, Á.M. Deep learning inference on edge: A preliminary device comparison. In *Proceedings of the International Conference on Intelligent Data Engineering and Automated Learning*; Springer: Berlin/Heidelberg, Germany, 2024; pp. 265–276.
11. Guyon, I.; Bennett, K.; Cawley, G.; Escalante, H.J.; Escalera, S.; Ho, T.K.; Macià, N.; Ray, B.; Saeed, M.; Statnikov, A.; et al. Design of the 2015 chlearn automl challenge. In *Proceedings of the 2015 International Joint Conference on Neural Networks (IJCNN)*; IEEE: New York, NY, USA, 2015; pp. 1–8.
12. Baratchi, M.; Wang, C.; Limmer, S.; Van Rijn, J.N.; Hoos, H.; Bäck, T.; Olhofer, M. Automated machine learning: Past, present and future. *Artif. Intell. Rev.* **2024**, *57*, 122. [\[CrossRef\]](#)
13. Schmitt, M. Automated machine learning: AI-driven decision making in business analytics. *Intell. Syst. Appl.* **2023**, *18*, 200188. [\[CrossRef\]](#)
14. Hutter, F.; Hoos, H.H.; Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the International Conference on Learning and Intelligent Optimization*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 507–523.
15. Thornton, C.; Hutter, F.; Hoos, H.H.; Leyton-Brown, K. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*; ACM: New York, NY, USA, 2013; pp. 847–855.
16. Witten, I.H.; Frank, E.; Hall, M.A.; Pal, C.J.; Data, M. Practical machine learning tools and techniques. In *Proceedings of the Data Mining*; Elsevier: Amsterdam, The Netherlands, 2005; Volume 2, pp. 403–413.
17. Kotthoff, L.; Thornton, C.; Hoos, H.H.; Hutter, F.; Leyton-Brown, K. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *J. Mach. Learn. Res.* **2017**, *18*, 1–5.
18. Salvador, M.M.; Budka, M.; Gabrys, B. Automatic composition and optimization of multicomponent predictive systems with an extended auto-WEKA. *IEEE Trans. Autom. Sci. Eng.* **2018**, *16*, 946–959. [\[CrossRef\]](#)
19. Kedziora, D.J.; Nguyen, T.D.; Musial, K.; Gabrys, B. On taking advantage of opportunistic meta-knowledge to reduce configuration spaces for automated machine learning. *Expert Syst. Appl.* **2024**, *239*, 122359. [\[CrossRef\]](#)
20. Feurer, M.; Klein, A.; Eggenberger, K.; Springenberg, J.; Blum, M.; Hutter, F. Efficient and robust automated machine learning. *Adv. Neural Inf. Process. Syst.* **2015**, *28*, 2962–2970.
21. Erickson, N.; Mueller, J.; Shirkov, A.; Zhang, H.; Larroy, P.; Li, M.; Smola, A. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv* **2020**, arXiv:2003.06505.
22. Feurer, M.; Eggenberger, K.; Falkner, S.; Lindauer, M.; Hutter, F. Auto-sklearn 2.0: Hands-free automl via meta-learning. *J. Mach. Learn. Res.* **2022**, *23*, 1–61.
23. Olson, R.S.; Moore, J.H. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Proceedings of the Workshop on Automatic Machine Learning*; PMLR: New York, NY, USA, 2016; pp. 66–74.
24. LeDell, E.; Poirier, S. H2o automl: Scalable automatic machine learning. In *Proceedings of the AutoML Workshop at ICML*; PMLR: New York, NY, USA, 2020; Volume 2020, p. 24.
25. Wang, C.; Wu, Q.; Weimer, M.; Zhu, E. Flaml: A fast and lightweight automl library. *Proc. Mach. Learn. Syst.* **2021**, *3*, 434–447.
26. Colangelo, P.; Segal, O.; Speicher, A.; Margala, M. AutoML for multilayer perceptron and FPGA co-design. In *Proceedings of the 2020 IEEE 33rd International System-on-Chip Conference (SOCC)*; IEEE: New York, NY, USA, 2020; pp. 265–266.
27. Jentzsch, F. Hardware-Aware AutoML for Exploration of Custom FPGA Accelerators for RadioML. In *Proceedings of the 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*; IEEE: New York, NY, USA, 2023; pp. 359–360. [\[CrossRef\]](#)
28. Achararit, P.; Hanif, M.A.; Putra, R.V.W.; Shafique, M.; Hara-Azumi, Y. APNAS: Accuracy-and-performance-aware neural architecture search for neural hardware accelerators. *IEEE Access* **2020**, *8*, 165319–165334. [\[CrossRef\]](#)
29. Liu, X.; Lu, G.; Lan, Z.; Shi, X. CoDAC: Algorithm-Hardware Co-Design by AutoML with Compression for Efficient Edge AI. In *Proceedings of the 2nd International Conference on Artificial Intelligence of Things and Computing*; ACM: New York, NY, USA, 2025; pp. 71–78.

30. Wang, K.; Liu, Z.; Lin, Y.; Lin, J.; Han, S. Hardware-centric automl for mixed-precision quantization. *Int. J. Comput. Vis.* **2020**, *128*, 2035–2048. [[CrossRef](#)]
31. Benmeziiane, H.; El Maghraoui, K.; Ouarnoughi, H.; Niar, S.; Wistuba, M.; Wang, N. A Comprehensive Survey on Hardware-Aware Neural Architecture Search. *arXiv* **2021**, arXiv:2101.09336.
32. Hutter, F.; Kotthoff, L.; Vanschoren, J. *Automated Machine Learning: Methods, Systems, Challenges*; Springer: Berlin/Heidelberg, Germany, 2019.
33. Grossman, R.; Bailey, S.; Ramu, A.; Malhi, B.; Hallstrom, P.; Pulleyn, I.; Qin, X. The management and mining of multiple predictive models using the predictive modeling markup language. *Inf. Softw. Technol.* **1999**, *41*, 589–595. [[CrossRef](#)]
34. Duarte, J.; Han, S.; Harris, P.; Jindariani, S.; Kreinar, E.; Kreis, B.; Ngadiuba, J.; Pierini, M.; Rivera, R.; Tran, N.; et al. Fast inference of deep neural networks in FPGAs for particle physics. *J. Instrum.* **2018**, *13*, P07027. [[CrossRef](#)]
35. Guazzelli, A.; Zeller, M.; Lin, W.-C.; Williams, G. PMML: An Open Standard for Sharing Models. *R J.* **2009**, *1*, 60–65. [[CrossRef](#)]
36. Li, X.; Lei, Y.; Jia, Z.; Zhang, Y.; Liu, H.; Chen, L.; Dong, W.; Li, S. Bridging Operator Semantic Inconsistencies: A Source-Level Cross-Framework Model Conversion Approach. *Proc. ACM Softw. Eng.* **2025**, *2*, 2030–2052. [[CrossRef](#)]
37. Hettiarachchi, D.L.N.; Davuluru, V.S.P.; Balster, E.J. Integer vs. floating-point processing on modern FPGA technology. In *Proceedings of the 2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*; IEEE: New York, NY, USA, 2020; pp. 0606–0612.
38. De Dinechin, F.; Detrey, J.; Cret, O.; Tudoran, R. When FPGAs are better at floating-point than microprocessors. In *FPGA '08: Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*; ACM: New York, NY, USA, 2008; p. 260. [[CrossRef](#)]
39. Xilinx, A. *Vitis High-Level Synthesis User Guide*; Technical Report; Amd Xilinx: Santa Clara, CA, USA, 2022.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.