UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

IAN KERSZ

Accuracy-Aware DNN Inference Provisioning for Heterogeneous Multi-FPGA Edge Systems

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering

Advisor: Prof. Dr. Antonio Carlos Schneider Beck Filho

Porto Alegre
July 2026

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profª. Marcia Cristina Bernardes Barbosa
Vice-Reitor: Prof. Pedro de Almeida Costa
Pró-Reitora de Graduação: Profª. Nádya Pesce da Silveira
Diretor do Instituto de Informática: Prof. Luciano Paschoal Gaspary
Coordenador do Curso de Engenharia de Computação: Prof. Paulo Francisco Butzen
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

“Intelligence is the ability to avoid doing work,
yet getting the work done.”
— Linus Torvalds

ACKNOWLEDGEMENTS

I am thankful to my parents for their unconditional love and for supporting me through all of these years. To Camila, for the love through all the days, nights, and weekends spent by my side while I worked, for always hearing me ramble on about my thoughts and helping me untangle them, and, more than anything, for making me happy.

My time at the university is one I will always look back on, thanks to my friends from PET Computação and UFRGS. I am grateful to the many who have passed through Lab 219, for the company, the conversations, and the work we built together, and to Arthur and Nathan, who shared most of this journey with me.

My thanks also go to the other professors of the research group, José Rodrigo Azambuja and Fernanda Lima Kastensmidt, for the opportunities they gave me and everything they taught me. To Michael G. Jordan, for his close guidance over the years, the countless discussions we shared, and for always making time for me. And finally to my advisor, Caco, for the constructive criticism and guidance that shaped me as a researcher, and for his support throughout it all.

ABSTRACT

The growing adoption of IoT devices at the network edge has established edge computing as an essential paradigm for meeting latency constraints in deep neural network (DNN) inference applications. Heterogeneous multi-FPGA platforms offer flexibility to accelerate these workloads, yet efficiently provisioning inference accelerators on such systems remains challenging due to the trade-off between accuracy, parallelism, and response time. High-accuracy accelerator variants consume more FPGA resources, limiting parallelism, while compact approximate variants increase throughput at the cost of inference quality. Varying inference demand intensifies this tension by creating contention that static provisioning approaches cannot accommodate. This work develops a provisioning framework that composes static accuracy filtering with dynamic contention-aware pool selection in a three-stage pipeline for heterogeneous multi-FPGA edge systems. The first stage applies a static accuracy filter that establishes a quality floor for the design space. The second stage employs an adaptive contention-aware window that dynamically adjusts the eligible design pool based on workload intensity. The third stage optimizes design-to-FPGA allocation and task-to-instance distribution through configurable provisioning strategies. The framework is evaluated under varying inference demand on clusters of 4–8 FPGAs comprising three board types. In simulation, a balanced configuration cuts response time 69% below the highest-accuracy baseline while staying 19 percentage points more accurate than the highest-throughput baseline.

Keywords: Edge Computing. Multi-FPGA Systems. DNN Inference. Provisioning. Accuracy. Response Time. Genetic Algorithm.

Provisionamento de Inferência de DNNs com Consciência de Acurácia para Sistemas de Borda Multi-FPGA Heterogêneos

RESUMO

A crescente adoção de dispositivos IoT na periferia da rede estabeleceu a computação de borda como um paradigma essencial para atender restrições de latência em aplicações de inferência baseadas em redes neurais profundas (DNNs). Plataformas multi-FPGA heterogêneas oferecem flexibilidade para acelerar essas cargas de trabalho, porém o provisionamento eficiente de aceleradores de inferência nesses sistemas permanece desafiador devido ao balanço entre acurácia, paralelismo e tempo de resposta. Variantes de aceleradores de alta acurácia consomem mais recursos de FPGA, limitando o paralelismo, enquanto variantes compactas aproximadas aumentam a vazão ao custo da qualidade de inferência. A demanda de inferência variável intensifica essa tensão ao criar contenção que abordagens estáticas não conseguem acomodar. Este trabalho desenvolve um sistema de provisionamento que compõe filtragem estática de acurácia com seleção dinâmica de conjunto ciente de contenção em um fluxo de três estágios para sistemas de borda multi-FPGA heterogêneos. O primeiro estágio aplica um filtro estático de acurácia que estabelece um piso de qualidade. O segundo estágio emprega uma janela adaptativa que ajusta dinamicamente o conjunto de aceleradores elegíveis com base na intensidade da carga de trabalho. O terceiro estágio otimiza a alocação de aceleradores a FPGAs e a distribuição de tarefas a instâncias através de estratégias configuráveis. O sistema é avaliado sob demanda de inferência variável em agrupamentos de 4 a 8 FPGAs abrangendo três tipos de placas. Em simulação, uma configuração equilibrada reduz o tempo de resposta em 69% em relação à referência de maior acurácia, com uma acurácia 19 pontos percentuais superior à da referência de maior vazão.

Palavras-chave: Computação de Borda. Sistemas Multi-FPGA. Inferência de DNNs. Provisionamento. Acurácia. Tempo de Resposta. Algoritmo Genético.

Contents

1 INTRODUCTION **15**

1.1 Chapter Organization **19**

2 BACKGROUND **21**

2.1 Edge Computing and IoT **21**

2.1.1 DNN Inference at the Edge 22

2.2 FPGAs as Edge Accelerators **22**

2.2.1 Heterogeneous Multi-FPGA Systems 23

2.3 Producing FPGA Designs **25**

2.3.1 Model-Level Optimization 25

2.3.2 Architecture-Level Synthesis 27

2.3.3 Design Space Exploration and the Design Pool 27

2.4 Provisioning and Resource Allocation **28**

2.5 Workload Modeling **29**

3 RELATED WORK **32**

3.1 Multi-Tenant FPGA Provisioning **32**

3.2 Multi-FPGA Scheduling and Management **33**

3.3 Runtime DNN Management and Adaptive Selection **33**

3.4 Positioning of This Work **34**

4 FRAMEWORK **35**

4.1 Static Time **36**

4.1.1 Design Pool Generation 36

4.2 Dynamic Time **37**

4.2.1 Stage 1: Static Accuracy Filter 38

4.2.2 Stage 2: Adaptive Contention-Aware Window 39

4.2.3 Stage 3: Provisioning Strategies 41

5 METHODOLOGY **44**
5.1 Hardware Platform **44**
5.2 Design Pool **45**
5.3 Workload Generation **46**
5.4 Evaluated Configurations **48**
5.4.1 Contention-Parameter Calibration 49
5.4.2 Contribution of the Pipeline Stages 50
5.5 Metrics **50**
5.6 Assumptions and Limitations **52**

6 RESULTS **53**
6.1 Provisioning Strategy Quality **53**
6.2 Calibrating the Adaptive Window **55**
6.3 Baselines and Featured Configurations **56**
6.4 Effect of Demand **58**
6.5 Heterogeneous Platform Scaling **61**
6.6 Contribution of the Pipeline Stages **63**

7 CONCLUSION **65**
7.1 Future Work **66**
7.2 Publications **66**

List of Figures

Figure 1.1 The edge computing layer between the cloud and the end devices. 15

Figure 1.2 End devices streaming inference requests to a multi-FPGA edge node. 17

Figure 1.3 Impact of Design Pool strategy on inference accuracy and response time in a multi-FPGA edge system. 18

Figure 2.1 Internal architecture of an FPGA. 24

Figure 2.2 Architecture of a multi-FPGA system. 24

Figure 2.3 Model-level optimization pipeline. 26

Figure 2.4 Demand gathered over the window T at three arrival rates. . . . 30

Figure 4.1 Architecture and operational workflow of AFLOAT. 35

Figure 4.2 The contention-aware window over a task type's accuracy-ranked designs. 40

Figure 5.1 The Design Pool across accuracy, throughput, and resource footprint. 47

Figure 5.2 Arrivals gathered over the window at the five evaluated demand levels. 48

Figure 6.1 GA and Greedy response time across the accuracy weight ω, by threshold τ. 53

Figure 6.2 GA and Greedy accuracy across the weight ω, one panel per threshold τ. 54

Figure 6.3 Response time and accuracy across the sensitivity s and window fraction f_w, per threshold τ. The chosen $s = 0.03$, $f_w = 0.50$ is boxed. 56

Figure 6.4 GA frontiers per threshold τ between the two baselines. The featured configurations are outlined in gold. 57

Figure 6.5 Response time and accuracy across demand D for the featured configurations and the two baselines. 58
Figure 6.6 Per-run response time and accuracy of the Fast configuration and the two baselines, colored by demand D. 59
Figure 6.7 Confidence regions of the featured configurations and the two baselines, each a 75% ellipse. 60
Figure 6.8 Pruning composition of the deployed designs per configuration, by hatching, across demand D. 60
Figure 6.9 Response time and accuracy across cluster size for the featured configurations and the two baselines, at peak demand. 62
Figure 6.10 Instances per board per configuration across cluster sizes, at peak demand. 62
Figure 6.11 Mean accuracy and response time of each pipeline configuration across demand D, markers darkening from $D = 16$ to 256. . . . 64

List of Tables

Table 5.1 FPGA devices of the simulated multi-FPGA cluster. 44
Table 5.2 Cluster configurations used in the evaluation. 45
Table 5.3 Design Pool characteristics by task type. 46

Table 6.1 Convergence time of the GA and Greedy strategies. 55
Table 6.2 Featured GA configurations and the two baselines. 57
Table 6.3 Makespan, energy, and convergence time across demand levels. . 61
Table 6.4 Makespan, energy, and convergence time across cluster sizes at peak demand. 63

LIST OF ABBREVIATIONS AND ACRONYMS

BRAM	Block RAM
CNN	Convolutional Neural Network
DNN	Deep Neural Network
DSA	Domain-Specific Architecture
DSE	Design Space Exploration
DSP	Digital Signal Processing
FF	Flip-Flop
FIFO	First-In, First-Out
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithm
IoT	Internet of Things
LUT	Look-Up Table
PE	Processing Element
pp	Percentage point
SIMD	Single Instruction, Multiple Data
XPE	Xilinx Power Estimator

LIST OF SYMBOLS

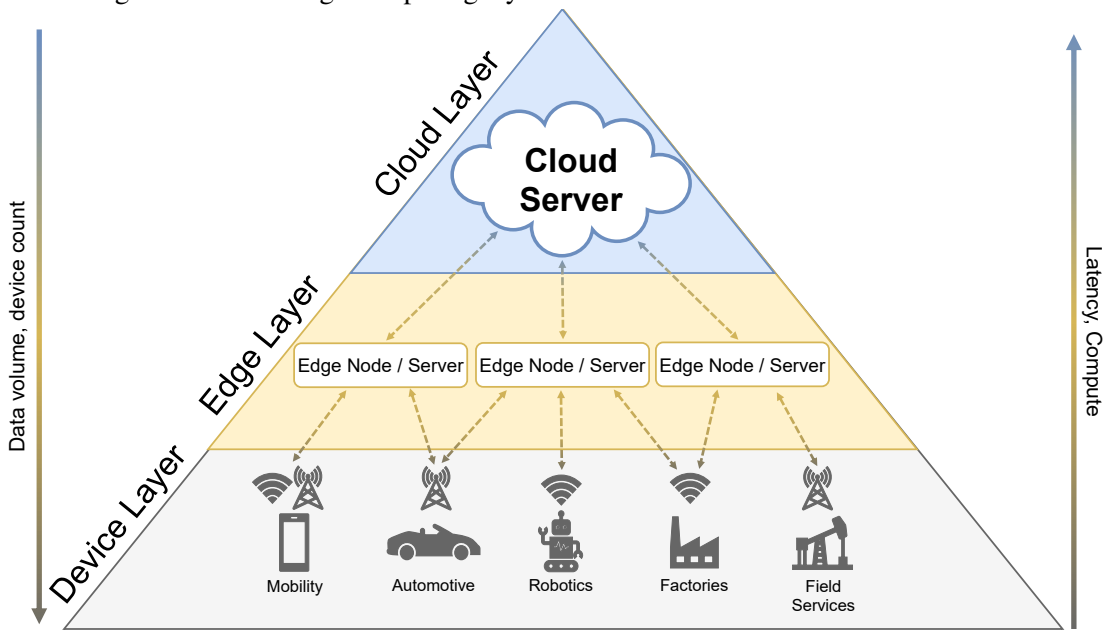
$\bar{a}_d$	Average Top-1 accuracy of a design
$\overline{Acc}$	Average Top-1 accuracy
$\mathcal{A}$	Active Pool
b	Excess load factor
c	Contention score
C	Cluster capacity
D	Workload demand
f_w	Window size fraction
F	Genetic algorithm fitness function
$\overline{g}$	Mean gather wait
G	Maximum generations
k	Window start index
n	Number of designs per task type
N	Number of tasks in a batch
p	Early-stop patience
P	Population size
$\mathcal{P}$	Design Pool
$\mathcal{P}_\tau$	Filtered Pool
R_{util}	Resource utilization fraction
$\overline{RT}$	Response time
s	Contention sensitivity parameter
S	Greedy heuristic score function
T	Batch window
T_{est}	Estimated execution time

w	Sliding window size
Δ_t	Execution-time range
λ	Arrival rate
μ	Mutation rate
ρ	Load factor
τ	Accuracy threshold for static filtering
ω	Accuracy importance exponent

1 INTRODUCTION

The amount of data generated outside of traditional data centers has grown to the point where transmitting every measurement to a centralized cloud is no longer practical. Driven by the proliferation of Internet of Things (IoT) devices, the center of data generation has shifted from centralized servers toward the network periphery, where sensors, cameras, and embedded systems continuously produce information that must be acted upon quickly (SHI et al., 2016). Edge computing has emerged as the paradigm that addresses this shift, placing computation closer to where data originates rather than relying on cloud infrastructure (JOUINI et al., 2024). As illustrated in Figure 1.1, the edge sits as an intermediate layer between the cloud, which offers abundant but distant compute, and the many end devices that generate the data. By keeping computation near the periphery, the edge avoids the latency and bandwidth cost of sending data to the cloud, while operating under far tighter resource constraints than a data center.

Figure 1.1 – The edge computing layer between the cloud and the end devices.



Source: adapted from Trigka & Dritsas (2025).

An increasing share of these edge workloads relies on Deep Neural Network (DNN) inference to extract actionable information from raw sensor inputs. Applications such as smart cameras for visual inspection and autonomous navigation systems must produce an inference result within strict time windows to drive a timely response, turning edge nodes from simple data aggregators into intelligent processing endpoints (SUBEDI et al., 2021a).

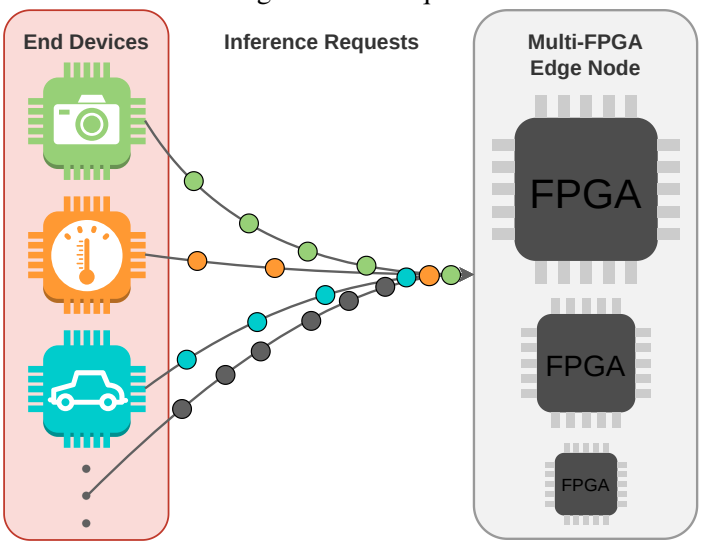
These workloads are also heterogeneous, relying on different DNN models that vary in structure and computational demand. Meeting their response-time targets is challenging because edge nodes must process continuous sensor streams with far less compute than a data center, and general-purpose processors are increasingly supplemented or replaced by dedicated hardware able to deliver higher throughput per watt (JIANG et al., 2020).

Field-Programmable Gate Arrays (FPGAs) present a compelling alternative for edge inference, as their internal logic can be reconfigured to implement custom hardware circuits. This allows a DNN model to be executed by a dedicated hardware accelerator, a circuit specialized for the computation of that particular model. Because such an accelerator can exploit fine-grained parallelism and operate at reduced numerical precision, it delivers high throughput per watt while occupying only a portion of the device's finite logic resources. Through a process called synthesis, an accelerator is converted into a design, a configuration that the FPGA can load. Placing a design onto an FPGA, or instantiating it, creates an instance. The same FPGA can be reconfigured to host different accelerator designs, so the hardware can be tailored to the application-specific performance and resource requirements of each workload, which is particularly valuable given the heterogeneity of edge DNN models (XU et al., 2020; KUON; ROSE, 2007).

A single FPGA device, however, often falls short of the computational capacity demanded by concurrent edge workloads. To overcome this bottleneck, multi-FPGA systems have been investigated as a practical scaling path, distributing accelerators across clusters of interconnected boards so that a provisioning layer can select which designs to instantiate and how to dispatch incoming tasks across the available resources (BIOOK-AGHAZADEH; RAVI; ZHAO, 2021a). A multi-FPGA node, the target of this work, is shown in Figure 1.2, where many end devices stream their inference requests for it to serve all at once. These clusters are rarely homogeneous, combining boards with different logic capacities, memory bandwidths, and power profiles, which makes the placement of these designs across them a non-trivial decision (XU et al., 2020).

Managing inference provisioning on such platforms introduces a challenging optimization problem. Typically, a multi-FPGA edge system maintains a library of pre-synthesized designs, collectively referred to as the Design Pool. From this pool, a provisioning layer makes two coupled decisions: which designs to instantiate on each FPGA, and how to distribute the incoming inference tasks among the resulting instances. Since a board can host only a limited number of instances at once, the first decision partitions the cluster's resources across the concurrent workloads, while the second balances the offered

Figure 1.2 – End devices streaming inference requests to a multi-FPGA edge node.



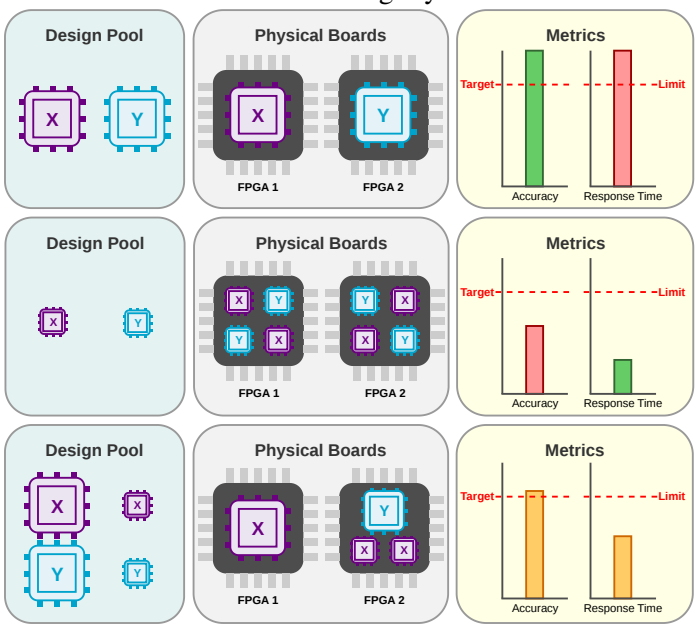
Source: the author.

load over the available instances. Together, these decisions determine the parallelism the cluster can achieve and, in turn, the response time experienced by each task.

Optimizing the DNN model itself adds accuracy as a further axis of the provisioning problem. Such model-level optimizations reduce a model's size and computational cost, producing several optimized versions of the model, each synthesized into a distinct design variant that trades inference accuracy for a smaller hardware footprint and faster execution (CHENG et al., 2018; GUO et al., 2019). High-accuracy variants generally consume more FPGA resources, limiting the number of instances that fit on a device and thereby restricting parallelism, whereas compact approximate variants occupy fewer resources and enable greater concurrency at the cost of inference quality. By selecting which variants to deploy, the provisioning layer now governs both response time and accuracy, placing the system at a specific point on the accuracy–response-time frontier.

Figure 1.3 illustrates this trade-off through three Design Pool strategies, each showing the pool contents, how designs are instantiated across the FPGA boards, and the resulting accuracy and response time relative to a target. When the pool is restricted to large, high-accuracy designs, only one instance fits per board, limiting parallelism and causing response time to exceed its limit under heavier load, even though accuracy stays above target. When the pool exposes compact variants, the provisioning layer packs many instances per board and keeps response time low, but accuracy consistently falls below target regardless of available slack. A mixed strategy that combines variants with complementary profiles can keep response time within its limit while holding accuracy near or above target, navigating the trade-off rather than being locked into either extreme.

Figure 1.3 – Impact of Design Pool strategy on inference accuracy and response time in a multi-FPGA edge system.



Source: the author.

This trade-off is further complicated by the varying demand of edge workloads. In real edge deployments, the volume of incoming requests rarely stays constant. A single physical event, such as motion detected across multiple surveillance cameras, can drive a sharp rise in concurrent requests (LI et al., 2025). As demand rises, these requests compete for the limited number of instances, creating contention that forces tasks to queue and increases response time. Under low demand, a provisioning strategy can afford to favor high-accuracy designs with minimal impact on response time, while under high demand the same strategy may need to use compact variants to sustain acceptable throughput. Keeping every design eligible would let the provisioning layer find the right mix, but the time spent searching the full pool adds to response time.

This work develops AFLOAT, a provisioning framework for DNN inference on heterogeneous multi-FPGA edge systems whose objective is to navigate the accuracy–response-time trade-off under varying demand rather than commit to a fixed operating point. AFLOAT addresses this through a three-stage pipeline. First, a static accuracy filter sets a quality floor, discarding designs that fall below it regardless of demand. Over the designs that remain, an adaptive window tracks the current demand, favoring the most accurate when it is low and shifting toward compact, faster ones as it rises. Finally, a provisioning strategy allocates these designs across the FPGAs and distributes the incoming tasks among the resulting instances, using a genetic algorithm or a greedy heuristic.

AFLOAT is evaluated through simulation on heterogeneous multi-FPGA clusters built from three FPGA devices: an Artix-7, a Zynq-7000, and a Kintex-7. These clusters serve requests for four DNN inference tasks, arriving at varying levels of demand. The evaluation spans 52,500 simulations across three accuracy thresholds, five demand levels, fourteen strategy configurations, and clusters ranging from four to eight FPGAs. Two baselines, one favoring accuracy and one favoring response time, bracket the trade-off across a further 5,000 simulations. A separate calibration tunes how the adaptive window responds to contention. Finally, a 1,000-simulation study isolates the contribution of each pipeline stage.

The main outcomes of this work are:

- A provisioning framework that composes static accuracy filtering with dynamic pool selection in a unified three-stage pipeline for DNN inference on heterogeneous multi-FPGA edge systems;
- An adaptive contention-aware window mechanism that adjusts the eligible Design Pool based on workload intensity;
- A simulation-based study that quantifies the potential benefits of AFLOAT across a range of accuracy thresholds, demand levels, cluster sizes, and provisioning strategies;
- A characterization showing that, in simulation, a balanced configuration of AFLOAT cuts response time by **69%** compared with the highest-accuracy baseline while staying **19 percentage points** more accurate than the highest-throughput baseline.

1.1 Chapter Organization

The remainder of this document is organized as follows. Chapter 2 presents the background concepts underlying this work, covering edge computing, FPGA-based acceleration, DNN model optimization, and workload modeling. Chapter 3 reviews related work in multi-tenant FPGA provisioning, multi-FPGA scheduling, and runtime DNN management, identifying the gap that this work addresses. Chapter 4 describes the AFLOAT framework architecture, detailing the three pipeline stages and the provisioning strategies. Chapter 5 presents the experimental methodology, including the simulation framework, hardware platform, workload generation, and evaluated configurations. Chapter 6 analyzes the experimental results, comparing the provisioning strategies and examining the

effect of demand, platform scaling, and the contribution of each pipeline stage. Chapter 7 summarizes the outcomes, discusses limitations, and outlines directions for future work.

2 BACKGROUND

This chapter introduces the concepts the rest of this work relies on, following the path a workload takes through the system. It begins with the edge environment where inference runs and the FPGAs that accelerate it, including the heterogeneous multi-FPGA platforms this work targets. It then focuses on the elements the framework works with directly: how the designs it provisions are produced, the provisioning decision it must make, and the varying demand it must track.

2.1 Edge Computing and IoT

Edge Computing moves data processing closer to where data is produced, at the edge of the network, rather than sending it to distant cloud servers. As the number of connected devices grows, sending all of their raw data to centralized servers becomes impractical, because of the added latency, the bandwidth it consumes, and privacy concerns (SHI et al., 2016). Processing data near its source instead reduces the traffic sent over the network and supports real-time decision-making. Unlike cloud data centers, which are built for high throughput with ample power and cooling, edge nodes operate under tight size, weight, and power limits, often with limited energy and passive cooling (JIANG et al., 2020).

Edge nodes are often expected to be multi-tenant, serving several independent applications on the same hardware at once. When resources are limited, a demanding task from one tenant can saturate the shared hardware and slow down the other applications running alongside it (ZOBAED et al., 2022a). This contention is made worse by the unpredictable arrival of IoT workloads, which often surge in response to physical events such as anomaly detections or synchronized sensor queries (BHATT; SHARMA, 2022), forcing the system to provision resources quickly enough to meet real-time deadlines. Rather than handle each task on its own, the system may gather the tasks arriving within a window into a batch and process them together.

These applications are also time-critical. A result that arrives after its deadline cannot influence the action it was meant to guide (CHEN et al., 2025). A node must therefore keep response quality high under normal load while still meeting deadlines when demand surges.

2.1.1 DNN Inference at the Edge

The edge workloads driving this demand have shifted from simple data aggregation to inference-driven processing. Modern edge applications increasingly rely on Deep Neural Network (DNN) models for tasks such as image classification, object detection, and anomaly recognition (JOUINI et al., 2024). A DNN is a trained function that maps an input, such as an image, to an output, such as a class label (GOODFELLOW et al., 2016). Inference is the use of such a model on new inputs, separate from the offline training that produced it. At the edge, inference runs repeatedly and under time pressure.

The dominant models for these perception tasks are Convolutional Neural Networks (CNNs), which slide small learned convolutional filters across an input to detect spatial patterns (LECUN et al., 1998). These filters are learned from many labeled examples during training. A CNN's computation is highly parallel and built from many multiply-accumulate operations, which general-purpose processors run inefficiently and with high energy cost (SUBEDI et al., 2021a).

A model's accuracy is the fraction of inputs it predicts correctly. A property of DNN inference that this work exploits is that this accuracy is not a fixed quantity: reducing a model's numerical precision and size lowers its computational cost in exchange for a controlled loss in accuracy. This flexibility, detailed in Section 2.3, lets a single inference task be served by accelerators at very different points in the cost-versus-quality space.

A single edge platform typically runs several of these applications at once. A smart-agriculture node, for example, might run crop-disease classification, pest detection, and yield estimation together, each with its own accuracy and throughput needs. The result is a workload mix the provisioning system must manage. Limited resources, multiple tenants, heterogeneous workloads, and varying demand together define the environment in which the framework developed in this work operates.

2.2 FPGAs as Edge Accelerators

Field-Programmable Gate Arrays (FPGAs) are integrated circuits whose hardware can be reconfigured after fabrication. This post-fabrication reconfigurability lets the circuitry be adapted to the needs of a particular workload. At the edge, a single FPGA can be reconfigured to accelerate different inference tasks as demands change (XU et al., 2020). The same flexibility makes deployed systems easy to update: swapping in a

different version of a model only requires re-synthesizing and reloading its design, not fabricating new hardware.

The custom circuits an FPGA builds also compute differently from general-purpose processors. Instead of executing a stream of instructions, an FPGA implements spatial computing, where data flows through a pipeline of arithmetic units laid out in space. This suits the dataflow-heavy computation of neural network inference, and avoids much of the energy that instruction-based processors spend fetching and decoding instructions. For real-time edge workloads, the result is more predictable latency and better energy efficiency (KUON; ROSE, 2007). Because the FPGA is reconfigurable, the same model can be mapped onto many different tailor-made architectures, each trading hardware resources for speed in a different way.

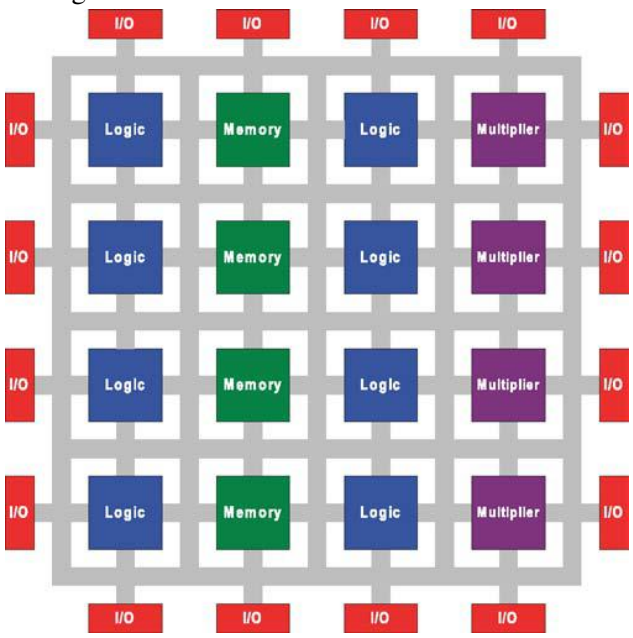
Such tailor-made architectures are known as Domain-Specific Architectures (DSAs), specialized for one class of problem rather than for general-purpose computing. A DSA optimizes its datapath, memory, and numerical precision for the target algorithm. On an FPGA this customization can reach the bit level, using very narrow integer types in place of 32-bit floating point to cut memory and arithmetic cost. The result is more computation packed into the same resources and power budget, which is what makes FPGAs attractive for neural network inference at the edge.

An FPGA's resources fall into a few categories. Look-Up Tables (LUTs) implement arbitrary combinational logic. Flip-Flops (FFs) hold values as they move through the pipeline. Digital Signal Processing (DSP) slices are dedicated arithmetic units for multiply-accumulate and related operations. Block RAMs (BRAMs) provide fast on-chip memory. Figure 2.1 shows how an FPGA arranges these resources. Programmable blocks of several types, here general-purpose logic alongside dedicated memory and multiplier blocks, are tiled across the device and linked by a programmable routing fabric, with programmable I/O blocks around the edge connecting the chip to the outside. How much of each resource a board provides, and how the routing combines them, determines which designs fit and how many instances can run on it at once, a constraint that directly shapes the provisioning decisions studied in this work.

2.2.1 Heterogeneous Multi-FPGA Systems

Because a single FPGA can host only a limited number of instances, larger workloads quickly outgrow it. Multi-FPGA systems address this by distributing accelerators

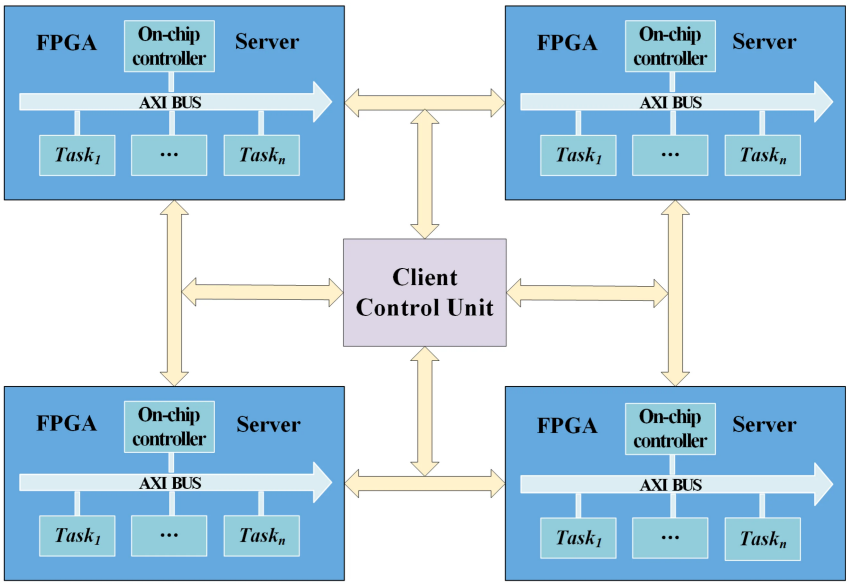
Figure 2.1 – Internal architecture of an FPGA.



Source: Kuon, Tessier & Rose (2008).

across several connected boards (BIOOKAGHAZADEH; RAVI; ZHAO, 2021a). Figure 2.2 shows one such arrangement: a central client control unit, commonly a host processor at the edge node, dispatches incoming tasks to several FPGA boards acting as servers, each running multiple tasks at once over its on-chip bus and returning the results to the controller. Coordinating the boards this way requires a provisioning layer, which decides which designs to load onto each board and how to dispatch incoming tasks to the resulting instances (BIOOKAGHAZADEH; RAVI; ZHAO, 2021b).

Figure 2.2 – Architecture of a multi-FPGA system.



Source: Li et al. (2025).

In a real deployment, a multi-FPGA cluster is rarely built from identical boards, since its hardware is often acquired at different times or chosen to balance cost against capability. Such a heterogeneous platform mixes boards of different capacities and architectures, from small low-power parts to larger devices with several times the logic, memory, and arithmetic resources (TANG et al., 2024). Because these devices differ in architecture, a design must be synthesized separately for each device type it runs on. Its resource footprint, however, stays roughly the same across them. A design that fits the smallest board therefore generally fits the larger ones too, and the cluster can be provisioned as a coherent whole.

A design's footprint decides not only whether it fits a board, but how many instances fit on one, and therefore how much parallelism is available. A large, high-accuracy design can take up most of a board, so only a single instance fits and parallelism is limited. A compact design has a smaller footprint, letting the provisioning layer pack several instances onto one board, which absorbs surges in demand at the cost of lower accuracy (YU et al., 2025). Larger boards widen these choices, holding either one large instance or many compact ones, so the balance differs across a heterogeneous cluster. Navigating this balance across the whole cluster is the provisioning problem this work addresses.

2.3 Producing FPGA Designs

The framework relies only on each design's characteristics, its resource use, throughput, and accuracy, so it can in principle provision designs from any accelerator architecture. Understanding the trade-offs it navigates among these characteristics requires knowing how the designs are produced. The following subsections describe that production, from optimizing a DNN model to synthesizing it into a configuration the FPGA can load, with this work's Design Pool detailed in Section 4.1.1 and Section 5.2.

2.3.1 Model-Level Optimization

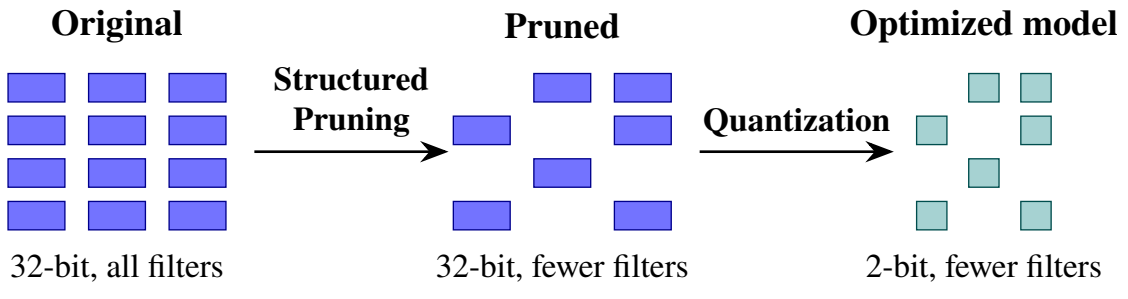
Model-level optimization reduces a neural network's computational cost before it is mapped to hardware, trading a controlled amount of accuracy for fewer operations on smaller data types. This work uses two complementary techniques.

Quantization reduces the numerical precision of a model's values, which can include both the weights and the activations passed between layers. Standard training works in 32-bit floating point, but many inference tasks tolerate far lower precision with little accuracy loss. Rather than rounding a trained model after the fact, quantization-aware training trains it at reduced precision from the start, letting the model adapt to the constraint and preserving accuracy down to very few bits (FRANCO; PAPPALARDO; FRASER, 2025). Low-precision integer arithmetic needs far fewer resources than floating-point or wider integer operations, so the resulting designs are more compact.

Structured pruning removes entire convolutional filters from a network, chosen by an importance measure (LI et al., 2017). A common strategy ranks filters by the L1-norm of their weights, the sum of their absolute values, and removes those with the smallest norms, assuming that low-magnitude filters contribute least to the output. Unlike unstructured pruning, which zeros individual weights and needs specialized sparse hardware, structured pruning removes whole filters at once. This shrinks the network directly, leaving a smaller, regular model that performs fewer operations and synthesizes efficiently without sparse-computation support.

Figure 2.3 shows the two techniques applied in sequence to a model. Structured pruning removes whole convolutional filters, so the network carries fewer of them, and quantization then narrows the data type of each remaining value, drawn as thinner and lighter blocks. Applying both in varying degrees yields optimized models spanning a wide range of accuracy and computational cost. The interaction is not simply additive: aggressive pruning of an already quantized model can cause disproportionate accuracy loss, because the reduced precision leaves fewer redundant parameters to absorb the removal. This non-linearity gives these models genuinely diverse profiles rather than a smooth continuum, which is what makes the resulting designs worth provisioning over.

Figure 2.3 – Model-level optimization pipeline.



Source: the author.

2.3.2 Architecture-Level Synthesis

Each optimized model must then be synthesized into a design the FPGA can load. This step introduces a second trade-off, now in hardware: devoting more circuitry to a layer makes it run faster but uses more resources, while reusing less circuitry saves resources at the cost of throughput. Sweeping this parameter lets a single model yield designs at many throughput and footprint points, whatever toolchain performs the synthesis.

This work performs the synthesis with FINN (UMUROGLU et al., 2017), an open-source framework from AMD Research, chosen for how easily it maps many different quantized and pruned models onto FPGAs. FINN builds streaming dataflow designs that give each layer its own hardware block and pipeline data through them, so throughput is set by the slowest block. Because it takes models straight from quantization-aware training, the precision choices of the previous subsection carry directly into the hardware.

FINN exposes this hardware parameter as folding, which sets how much of each layer runs in parallel. A layer's block carries out the model's multiply-accumulate operations on the FPGA's LUTs and DSP slices, and keeps its parameters in BRAM. Folding sets how many of these run at once, through two parameters: SIMD for the input dimension and PE for the output dimension. Higher values put more of this logic to work, raising throughput but consuming more resources, while lower values reuse fewer units to save resources at the cost of throughput.

A single model can be folded in many different ways, each producing a distinct design. Together with the model-level optimizations of the previous subsection, this produces designs that differ along three axes at once: hardware footprint, throughput, and accuracy. Searching this space for a useful and diverse set of designs calls for a systematic exploration.

2.3.3 Design Space Exploration and the Design Pool

Design space exploration (DSE) is the systematic search of a design space for implementations that balance competing objectives. For an FPGA accelerator, that space is large, spanning model-level choices such as precision and pruning and architecture-level choices such as how much parallelism to give each layer, with each combination offering a different mix of accuracy, throughput, and resource use. DSE generates and characterizes points in this space so that good designs can be identified rather than hand-

picked. Parts of this search can be automated: FINN-R (BLOTT et al., 2018) explores folding configurations to hit a target throughput.

Exploring both kinds of choice, the model-level optimizations described earlier and FINN folding configurations chosen to fit the target FPGA's resources (ELY et al., 2025b), yields a Design Pool: a library of designs with diverse accuracy, resource, and throughput characteristics from which the framework selects at runtime. The Design Pool is the raw material for every provisioning decision the framework makes. Chapter 5 describes the specific Design Pool used in this work's evaluation, including the models, datasets, and resulting design counts.

2.4 Provisioning and Resource Allocation

With a heterogeneous cluster on one side and a Design Pool on the other, the system must decide how to bring the two together. This decision is provisioning, and it rests on two coupled choices. Design allocation selects which designs to instantiate and on which boards, subject to each board's resource capacity. Task distribution assigns each incoming inference request to a compatible instance. The two are interdependent: the instances allocation creates are the only targets distribution has, and creating too few of them leaves tasks waiting in the queue.

Provisioning is naturally framed as a constrained optimization problem. The objective is a performance measure such as the total time to complete a batch of tasks, or the response time across tasks. The constraints are the finite resource capacity of each board and any minimum-accuracy requirement imposed on the served tasks. Even stated this plainly, the problem is combinatorial. A batch of tasks, a cluster of boards, and a Design Pool together define a space of candidate allocations that grows explosively with each dimension. Problems of this form, assigning and scheduling tasks onto bounded resources, are NP-hard (GAREY; JOHNSON, 1979; ZENG et al., 2023), so computing a provably optimal allocation within the time budget of a real workload is impractical. Practical systems therefore rely on strategies that converge to good solutions quickly rather than searching exhaustively.

A greedy heuristic builds an allocation incrementally, scoring the candidates and making the locally best choice at each step, for example placing the design that most improves the objective given the resources still available (CORMEN et al., 2022). Greedy strategies are fast and deterministic, which makes them attractive when decision latency

is itself part of the response time. Their limitation is that locally optimal choices do not always add up to a globally good allocation, because early placements can consume resources that later tasks need.

A metaheuristic trades some of this speed for a broader exploration of the search space. Genetic Algorithms (GAs) are a widely used family of metaheuristics inspired by natural selection (HOLLAND, 1992). A GA maintains a population of candidate solutions, each scored by a fitness function that encodes the objective. It evolves the population over generations through three operators: *selection* favors fitter candidates as parents, *crossover* recombines two parents into new candidates, and *mutation* randomly perturbs a candidate to add diversity (GOLDBERG, 1989). This broad exploration lets a GA escape the local optima that can trap a greedy heuristic, but it evaluates many candidates, which makes it slower. When the objective must balance competing goals such as response time against accuracy, a weighted fitness function can encode both directly, allowing the evolutionary search to navigate this trade-off.

The framework draws on both of these approaches, the greedy heuristic and the genetic algorithm, detailed in Chapter 4.

2.5 Workload Modeling

A workload's tasks do not arrive evenly. The volume of requests reaching the platform rises and falls over time, and that variation drives how much contention the system faces. Modeling it calls for a way to describe how requests arrive and a way to measure the load they place on the available resources.

Tasks arrive at random times, set by the applications that generate them. In IoT and edge settings, these arrivals are commonly described by a Poisson process, in which requests occur independently of one another at a constant average rate λ (RUIZ-GUIROLA; LÓPEZ; MONTEJO-SANCHEZ, 2025). Its defining property is that arrivals are memoryless: the time until the next request does not depend on when the previous ones arrived.

Rather than being handled individually, requests arriving within a fixed window of length T are gathered into a batch and provisioned together (Section 2.1). The number of requests collected over that window is the demand D, the quantity the provisioning layer must serve in one round. For Poisson arrivals at rate λ, the demand gathered over a window of length T is, on average,

$$D = \lambda \times T, \tag{2.1}$$

so a faster arrival rate or a longer window raises the demand a batch carries. A surge in activity may shift both the arrival rate and the exact timing of the arrivals, but only the rate changes how many land in a window. What reaches the provisioning layer is then the demand D gathered over the window, not the spacing of the arrivals that produced it.

Figure 2.4 shows this for three arrival rates over a single window T. On the left, each row gives the distribution of times between consecutive requests: since the arrivals are memoryless, a higher rate λ concentrates these gaps nearer zero, so requests follow one another more closely. On the right, those arrivals fill the window, drawn as individual requests beneath the rate they form, on a shared scale so that a heavy demand towers over a light one. A higher rate packs more requests into the same window, so the demand D grows with it, as Equation 2.1 predicts. The arrival rate alone therefore sets the demand level, from light to heavy. Chapter 5 gives the demand levels used in the evaluation.

Figure 2.4 – Demand gathered over the window T at three arrival rates.



Source: the author.

Demand on its own does not determine how much contention the system faces. What matters is demand measured against the resources available to serve it. A system has a finite capacity C, the number of requests it can serve within the window before they begin to compete for the same resources. The ratio of the two is the load factor ρ (KLEINROCK, 1975):

$$\rho = \frac{D}{C} \tag{2.2}$$

When $\rho < 1$, capacity exceeds demand and most requests are served without waiting, so contention stays low. When ρ rises to 1 and beyond, demand matches and then exceeds what the system can serve at once. Requests then queue for busy resources, and response time climbs. The load factor therefore folds the arrival side and the capacity side into a single normalized measure of how loaded the system is. A value near or above 1 signals that the workload is straining the available resources. The framework uses this load factor to gauge contention at runtime, detailed in Chapter 4.

3 RELATED WORK

The framework developed in this work draws on three lines of research, each addressing part of the provisioning problem. The first looks at sharing a single FPGA among competing workloads, where only a few designs fit on one device. The second moves to clusters of boards, where each design must also be placed across the available devices. The third manages DNN accelerators at runtime, adapting which designs are eligible as the workload changes. The chapter closes by placing this work alongside these efforts and pointing to the gap it fills.

3.1 Multi-Tenant FPGA Provisioning

The first scale of provisioning is a single FPGA shared among competing workloads. For cloud systems, Mbongue et al. (2020) partition one FPGA into isolated regions with a network-on-chip, each region acting as an independent virtual accelerator and raising overall utilization. Minhas et al. (2021) share the area more dynamically, partitioning tasks across the device and picking among accelerator profiles with different resource and throughput trade-offs as the workload shifts. Nguyen & Kumar (2020) pursue the same goal offline, using design space exploration to arrange the FPGA's reconfigurable regions so it can host as many tenant accelerators as possible.

Other approaches coordinate how those workloads run. Zeng et al. (2023) co-explore the accelerator architecture, scheduling, and mapping needed to serve several DNNs at once, computing the assignment offline. THEMIS (KARABULUT et al., 2025) instead targets fairness, scheduling competing tenants for balanced access to the device under timing and energy budgets.

These works lay the foundations of FPGA provisioning. Yet each acts within a single device, optimizing goals such as utilization, fairness, or energy while holding the accuracy of each design fixed.

3.2 Multi-FPGA Scheduling and Management

As workloads grow beyond a single device, multi-FPGA systems distribute accelerators across a cluster of connected boards. Biookaghazadeh, Ravi & Zhao (2021a) split a single large CNN across these boards, balancing the pipeline to reach near-linear throughput. Their focus is model parallelism for one large network rather than provisioning many independent inference tasks.

Scheduling many tasks across the cluster has been approached in several ways. Razavi et al. (2022) exploit recurring patterns in DNN requests, staggering their dispatch to reduce contention. Clementi (YU et al., 2025) balances load across boards and overlaps communication with computation, though for graph processing in the cloud rather than inference at the edge. Li et al. (2025) schedule under resource and reliability constraints, splitting large tasks into subtasks and ordering them ahead of time.

CHEF (TANG et al., 2024) deploys different DNN models on a heterogeneous FPGA cluster, choosing which accelerators go on which boards and which accelerator each model runs on. These choices, though, are made offline, before any requests arrive.

These works scale provisioning from one FPGA to a whole cluster, coordinating where designs run and how tasks reach them. Yet across all of them the set of available designs is fixed: they schedule and allocate over it, but never revisit it as the workload shifts.

3.3 Runtime DNN Management and Adaptive Selection

Beyond scheduling, systems select and adapt accelerator designs at runtime. AdaFlow (KOROL et al., 2022) builds a library of CNN variants at different pruning levels on FINN and, at runtime, selects which one to run, guided by the current workload and a configurable accuracy threshold. Its scope is a single design rather than a pool of designs provisioned together.

Sharing the hardware among tenants takes several forms. Planaria (GHODRATI et al., 2020) splits a systolic array into smaller engines so several DNNs run side by side. AuRORA (KIM et al., 2023) virtualizes the accelerators, binding each workload to whichever is free to keep tenants served fairly. On edge servers, model managers (SUBEDI et al., 2021b; ZOBAED et al., 2022b) juggle which models stay resident in limited memory to keep latency-sensitive applications responsive.

The designs these systems run must first be produced by design space exploration. DNNExplorer (ZHANG et al., 2020) explores accelerator architectures, DeepAxe (TAHERI et al., 2023) approximation and reliability, FPGA-QNN (TASCI et al., 2025) quantization and folding on FINN, and APPQ-CNN (ZHANG et al., 2024) pruning and quantization, each yielding designs at different points in the accuracy, resource, and throughput space. What they generate is exactly the pool a provisioning layer needs, but choosing among it at runtime is left to that layer. AdaPEx (KOROL et al., 2023) adapts during inference on a single accelerator, co-optimizing pruning with early exit to skip the rest of the network once an input is easy enough to classify.

Across these runtime and adaptive approaches, adaptation operates at the per-model or per-instance level rather than at the platform level. That leaves room to adapt across an entire multi-FPGA cluster, coordinating the many designs it runs at once.

3.4 Positioning of This Work

The work reviewed here addresses the problem in pieces. Provisioning has been studied on a single FPGA and scaled across a cluster, while designs have been selected and adapted at runtime, even trading accuracy for speed. Combining them could prove beneficial.

In particular, existing multi-FPGA provisioning works either operate with static design selections determined offline or adapt designs without considering accuracy as an explicit trade-off dimension. Accuracy-aware approaches such as AdaFlow and AdaPEx adapt at the per-instance level but do not extend their adaptation to platform-wide design pool management. Adapting for accuracy across a whole multi-FPGA cluster has not been done.

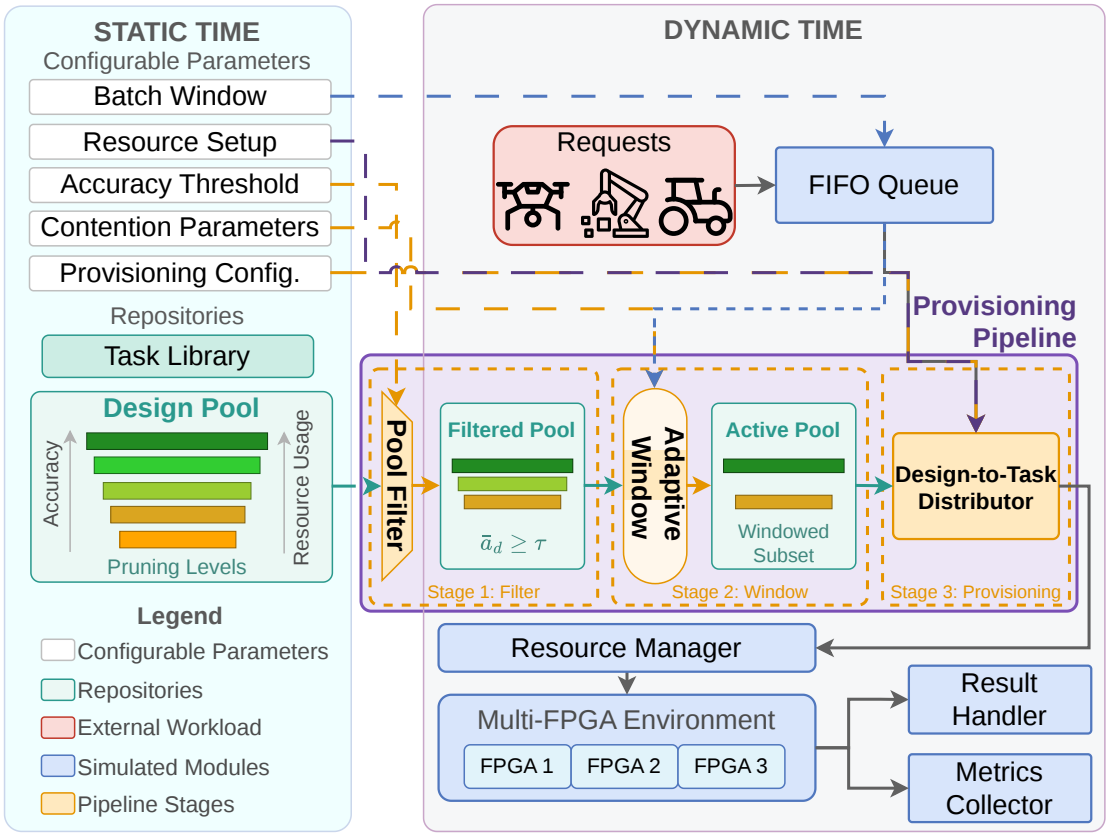
The framework developed in this work addresses precisely this, through a three-stage pipeline. A static accuracy filter sets a quality floor, independent of the workload, while an adaptive window shifts the eligible designs as contention rises. A provisioning strategy then allocates these designs across the cluster. Together, they keep accuracy high under light load and fall back on compact designs to protect response time when demand surges.

4 FRAMEWORK

AFLOAT is the provisioning framework developed in this work, running on a heterogeneous multi-FPGA cluster that serves DNN inference requests as they arrive in batches. For each batch, it resolves the two coupled provisioning decisions of design allocation and task distribution: which designs to instantiate on each board, and how to distribute the requests among the resulting instances. Its objective is to navigate the accuracy–response-time trade-off under varying demand, holding accuracy high while load is light and turning to compact, faster designs as contention rises.

Figure 4.1 shows the architecture of AFLOAT, split into two phases. *Static Time* configures the system before any requests arrive, setting the framework's parameters and assembling the repositories that provisioning later draws on. *Dynamic Time* then handles the incoming requests, gathering them into batches and running a provisioning cycle for each one. The chapter is organized around these two phases: Section 4.1 covers Static Time, including how the Design Pool is generated, while Section 4.2 covers Dynamic Time, walking the three pipeline stages that turn each gathered batch into an allocation.

Figure 4.1 – Architecture and operational workflow of AFLOAT.



Source: the author.

4.1 Static Time

During *Static Time*, before it serves any requests, the framework exposes a group of parameters and populates two repositories that *Dynamic Time* relies on.

Two parameters describe the deployment itself. The Batch Window T sets how long requests are gathered before a provisioning cycle begins, and with it the demand each cycle provisions for. The Resource Setup describes the cluster, listing its devices and the resource budget of each in LUTs, FFs, DSPs, and BRAMs.

Three further parameters tune the stages of the pipeline. The Accuracy Threshold τ sets the minimum accuracy a design must reach to pass the static accuracy filter (Section 4.2.1). The Contention Parameters, a sensitivity s and a window fraction f_w, govern how the adaptive window responds to contention (Section 4.2.2). The Provisioning Configuration selects the provisioning strategy and its settings, such as the accuracy importance weight ω (Section 4.2.3).

Two repositories are populated alongside these parameters. The Task Library lists the application types the system supports, each a DNN inference task with its own model and dataset, and maps every incoming request to one of them. The Design Pool holds the pre-synthesized designs that execute those tasks. For each task type it provides several design variants, which can differ in pruning ratio, quantization precision, and hardware parallelism, giving them distinct resource footprints, throughput, and accuracy.

4.1.1 Design Pool Generation

The Design Pool is not generated by AFLOAT, but by a separate design-space exploration carried out in prior work, and is reused here as a fixed input. This exploration builds the pool offline in two steps. The first step optimizes each DNN model with the two model-level techniques introduced in Section 2.3.1, quantization and structured pruning. Quantization-aware training with Brevitas (FRANCO; PAPPALARDO; FRASER, 2025) reduces every weight and activation to 2-bit precision. At this precision the models stay competitive in accuracy on the target tasks, while each design's resource footprint drops sharply, so a single board can hold many more instances. Structured pruning then removes convolutional filters at six rates, from none up to 90%, following the methodology of Li et al. (2017), and each pruned model is fine-tuned for 40 epochs to recover the accuracy lost with the removed filters. Together these yield several model variants per task type, from

the most accurate to the most compact.

The second step maps each model variant onto FPGA hardware with FINN (UMUROGLU et al., 2017), which gives every network layer its own streaming dataflow block (Section 2.3.2). Folding sets how much hardware each layer receives, trading resource use for throughput, so one model variant can be built at several throughput-resource points. Following the folding-exploration methodology of HARA (ELY et al., 2025b), this parameter is swept to produce five builds per variant, spanning the lowest to the highest throughput the variant can reach. Together, the two steps yield a pool of design variants spanning wide ranges of accuracy, resource use, and throughput.

Each build is characterized once it is generated. Vivado (VIVADO. . . , 2023) reports its resource use from the post-implementation results. Using these resource values, the Xilinx Power Estimator (XPE) estimates the dynamic power the design consumes. Its accuracy is that of its model variant, measured on a test set. The composition of the resulting pool is reported in Section 5.2.

4.2 Dynamic Time

The *Dynamic Time* phase governs the continuous operation of the system. Inference requests arrive from external sources and are buffered in the FIFO Queue. Once requests have accumulated for the configured Batch Window T, a provisioning cycle begins over the gathered batch. Because the batch is complete before the cycle starts, the window T trades how long requests wait to be gathered against how much demand each cycle optimizes over.

The provisioning cycle runs three pipeline stages that together make up AFLOAT, grouped together in Figure 4.1. They are implemented in Rust for the performance that per-batch provisioning demands. Before the provisioning strategy runs, the Design Pool is narrowed to the designs eligible for the current batch. The Pool Filter applies the static accuracy filter once, dropping every design below the Accuracy Threshold to leave a Filtered Pool (Section 4.2.1). For each batch, the Adaptive Window then trims the Filtered Pool to the designs suited to the batch's contention, forming the Active Pool (Section 4.2.2). The Design-to-Task Distributor runs the provisioning strategy over the Active Pool, resolving the coupled decisions of design allocation and task distribution (Section 4.2.3).

Every other module shown belongs to the simulator, a Python program developed in this work. It generates the workloads and scenarios that drive the framework and stands in for the physical cluster its allocations would run on. Python keeps assembling and varying these experiments straightforward. The Resource Manager takes the allocation the strategy produces and checks that the designs fit the boards in the Multi-FPGA Environment. The Result Handler then verifies that every task is assigned to a compatible design, and the Metrics Collector records the response times, device occupancy, and accuracy the evaluation reports.

4.2.1 Stage 1: Static Accuracy Filter

The first stage sets a quality floor on the Design Pool. Because its only inputs, the Design Pool and the accuracy threshold, do not change between batches, the framework computes the Filtered Pool once and caches it, reusing it for every provisioning cycle.

Not every design in the pool suits every deployment. Where inference quality matters, a compact but inaccurate design should never be provisioned, however fast it is. The accuracy threshold τ rules these designs out. For each design d, the filter takes its average Top-1 accuracy $\bar{a}_d$ across the task types it supports and keeps the design only if it meets the threshold:

$$\bar{a}_d \geq \tau \tag{4.1}$$

The Filtered Pool $\mathcal{P}_\tau$ collects the designs that pass, together with a coverage set $\mathcal{P}_{\text{cov}}$ that keeps every task type executable:

$$\mathcal{P}_\tau = \{d \in \mathcal{P} : \bar{a}_d \geq \tau\} \cup \mathcal{P}_{\text{cov}} \tag{4.2}$$

A high threshold could remove every design for some task type, leaving it unexecutable. The coverage set $\mathcal{P}_{\text{cov}}$ prevents this by restoring, for any such task type, its single highest-accuracy design. Every task type therefore keeps at least one design, whatever the threshold.

Raising τ shrinks the Filtered Pool. At $\tau = 0$ no design is removed. A moderate threshold of 50% or 75% removes the least accurate designs but keeps many compact ones. A strict threshold of 90% leaves only the most accurate few, sharply reducing the eligible designs.

This narrowing serves two purposes. First, no task can run on a design below the threshold, apart from those kept by the coverage set. Second, the smaller pool leaves the later stages less to search, which can shorten provisioning.

4.2.2 Stage 2: Adaptive Contention-Aware Window

The second stage adjusts the eligible designs to each batch's workload. The static accuracy filter sets a quality floor, but it cannot react to how heavily a given batch loads the system. When high demand drives up contention, compact and faster designs become worth their lower accuracy, because they let more instances run in parallel and absorb the load. When demand is light, the system can hold out for the most accurate designs. The Adaptive Window captures this by reading the batch's demand, estimating its contention, and narrowing the Filtered Pool to a matching subset, the Active Pool.

Contention is read from the batch's load factor $\rho = D/C$, the demand-to-capacity ratio defined in Section 2.5. Here the demand D is the number of requests in the batch, and the capacity C is the number of instances of the most accurate design that fit on the cluster. A load factor of $\rho \leq 1$ means the cluster could serve the whole batch at full accuracy and signals little pressure, while $\rho > 1$ means the demand outstrips that capacity. The Adaptive Window isolates the excess above one:

$$b = \max(\rho - 1,\ 0) \tag{4.3}$$

and scales it by a sensitivity s into a contention score $c \in [0, 1]$:

$$c = \min(b \cdot s,\ 1) \tag{4.4}$$

A higher s treats milder overloads as contention, while a lower s waits for heavier ones. The score reaches $c = 1$ when demand far exceeds capacity, and falls to zero whenever the cluster can serve the batch at full accuracy.

The score then positions a window over the designs of each task type, ranked by descending accuracy. The window spans a configurable fraction f_w of that type's n designs:

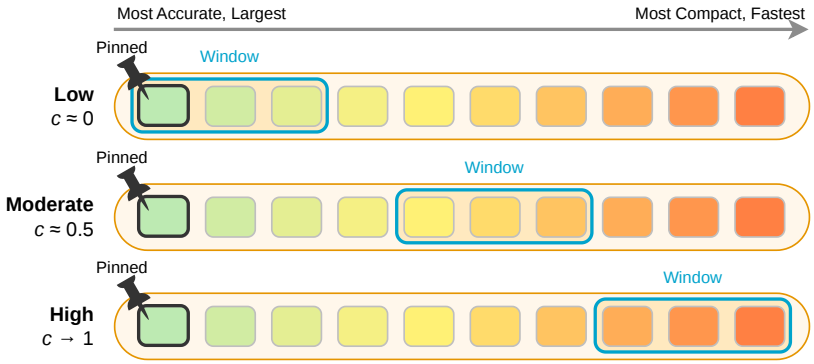
$$w = \lceil n \cdot f_w \rceil \tag{4.5}$$

Its start index follows the contention score:

$$k = \text{round}(c \cdot (n - w)) \tag{4.6}$$

At low contention the window sits at the top of the ranking, exposing only the most accurate designs, and as contention rises it slides toward the compact, faster ones at the bottom, as Figure 4.2 shows.

Figure 4.2 – The contention-aware window over a task type's accuracy-ranked designs.



Source: the author.

As in the static filter, a guarantee keeps a high-accuracy option always within reach. A configurable number of the highest-accuracy designs per task type are pinned to the Active Pool regardless of the window position, so the provisioning strategy keeps at least one high-accuracy design for each type even under heavy contention. The Active Pool is then the union of these pinned designs and the windowed ones:

$$\mathcal{A} = \mathcal{P}_{\text{pinned}} \cup \{d_k, d_{k+1}, \ldots, d_{k+w-1}\} \tag{4.7}$$

The static filter and this adaptive window are not redundant, because they address orthogonal concerns. On its own, the static filter cannot adapt. Set high, it removes the compact designs needed to absorb high demand. At light load, it cannot turn the spare capacity into higher accuracy. The adaptive window adds that missing adaptation, but on its own it has no floor, sliding under heavy contention onto designs below any acceptable accuracy. Composing the two holds the quality floor while climbing toward accuracy when contention is low and falling back toward compact designs when it is high.

4.2.3 Stage 3: Provisioning Strategies

The provisioning strategy turns the Active Pool into the final allocation, resolving the two coupled decisions of design allocation and task distribution (Section 2.4): which designs to instantiate on each board, and which instance runs each task. The framework keeps this strategy modular, so any method that maps the batch's tasks onto instances of the Active Pool's designs can be added without changing the rest of the pipeline. This work implements two that differ sharply in how they search that space: a greedy heuristic (Greedy) that builds one allocation in a single deterministic pass, and a genetic algorithm (GA) that searches many allocations for a better one at a higher compute cost.

The greedy heuristic is a lightweight, fast-converging strategy, suited to settings where the provisioning decision must itself be made quickly. Operating on the batch of tasks *batch*, the available boards *fpgas*, and the Active Pool *pool*, it constructs an *allocation* that maps each task to a design instance on a board, in the single pass summarized in Algorithm 4.1.

Algorithm 4.1 Greedy provisioning.

```
1:  allocation ← ∅
2:  sort batch by descending execution-time range Δ_t        ▷ prioritization
3:  for all task type in batch do                            ▷ coverage
4:      d ← design of this type in pool minimizing S
5:      f ← first board in fpgas that fits d
6:      add instance (d, f) to allocation
7:  end for
8:  for all remaining task t in batch, in priority order do  ▷ assignment
9:      reuse ← instances in allocation of t's type
10:     new ← designs of t's type that fit a board in fpgas
11:     (d, f) ← option in reuse ∪ new minimizing S
12:     assign t to (d, f), adding the instance to allocation if new
13: end for
14: return allocation
```

Every candidate task-design pairing is scored by

$$S = \frac{T_{\text{est}} / R_{\text{util}}}{\text{Acc}^{\omega}} \tag{4.8}$$

where T_{est} is the design's estimated execution time, the number of frames the request carries divided by the design's characterized throughput, R_{util} the fraction of a board's resources it consumes, and Acc its Top-1 accuracy. Minimizing S favors designs that are fast and resource-light, and the exponent ω sets how strongly low accuracy is penalized.

The heuristic first orders *batch* by each task's execution-time range Δ_t, the spread between its slowest and fastest design in *pool*, and places the widest-range tasks first. Handling the most design-sensitive tasks first, as workflow and load-balancing schedulers do (DJIGAL et al., 2021; MAO; CHEN; LI, 2014), matters because early placements consume the boards' finite resources, and these tasks gain the most from choosing their hardware while resources are still abundant.

A coverage pass then instantiates one design for each task type in *batch*, taking the design in *pool* that minimizes S on the first board in *fpgas* with room for it. This keeps every task type executable, rather than letting earlier types exhaust the resources and strand a later one.

A final pass assigns the remaining tasks in priority order. For each task it weighs reusing an instance already in *allocation* against opening a new one on a board with spare resources, and takes whichever minimizes S. The completed *allocation* is the heuristic's output.

The genetic algorithm trades the greedy heuristic's speed for a broader search of the allocation space. Following an approach used for related provisioning and accelerator-mapping problems (JORDAN et al., 2023; KAO et al., 2022), it evolves a population of candidate allocations over many generations, escaping the local optima that can trap a single-pass heuristic (Section 2.4) at the cost of evaluating far more of them. Algorithm 4.2 summarizes the procedure.

Algorithm 4.2 Genetic provisioning.

$pop \leftarrow P$ random individuals
$best \leftarrow \emptyset$, $stall \leftarrow 0$
for $g = 1$ to G **do**
    decode and score every individual by F ▷ infeasible $\Rightarrow F = \infty$
    $fittest \leftarrow$ individual with the lowest F
    if $best = \emptyset$ **or** $F(fittest) < F(best)$ **then**
        $best \leftarrow fittest$, $stall \leftarrow 0$
    else
        $stall \leftarrow stall + 1$
    end if
    if $stall \geq p$ **then**
        break ▷ early exit
    end if
    $parents \leftarrow$ selection over pop
    $pop \leftarrow \{best\} \cup \text{mutate}(\text{crossover}(parents), \mu)$
end for
return decoded allocation of *best*

Each individual I encodes a candidate allocation as a sequence of triples that assigns every task in *batch* to a board and a compatible design from the Active Pool:

$$I = \{(t_1, f_1, d_1),\ (t_2, f_2, d_2),\ \ldots,\ (t_N, f_N, d_N)\} \tag{4.9}$$

where t_i is the task, f_i its board, and d_i a design of the task's type. The initial population holds P such individuals, with every task's board and design drawn at random.

An individual only proposes an assignment, so it is decoded into a concrete allocation before it can be scored. Each task type is covered first, then every remaining task is placed on the instance that finishes it soonest, opening a new instance only when that both fits and completes earlier than reusing an existing one. An individual is infeasible when a design does not match its task's type, a task type is left uncovered, or the boards' resources run out before every task is placed. Such individuals are assigned fitness ∞ and do not survive selection.

Each feasible allocation is scored by a fitness function F that divides the batch's response time by the accuracy of its chosen designs:

$$F = \frac{\overline{RT}}{\left(\frac{1}{N}\sum_{i=1}^{N} \mathrm{Acc}(d_i)\right)^{\omega}} \tag{4.10}$$

where $\overline{RT}$ is the response time and the denominator its average Top-1 accuracy, raised to the same exponent ω as the greedy score S. With $\omega = 0$ the search minimizes response time alone, and larger ω steers it toward more accurate designs at the cost of response time. Because the earlier stages have already removed unacceptable and ill-suited designs, F need only balance response time against accuracy within the Active Pool.

Each generation evolves the population through the operators introduced in Section 2.4. Tournament selection takes the fittest of a few random individuals as each parent, one-point crossover swaps the tails of two parents' assignment sequences, and mutation reassigns a single task to a different board and compatible design with probability μ. Elitism carries the best individual *best* unchanged into the next generation, so the best allocation found is never lost.

The search runs for at most G generations, stopping early once the best fitness has not improved for p consecutive generations. Its result is the decoded allocation of *best*, the fittest individual seen. The population size P, generation cap G, mutation rate μ, patience p, tournament size, elitism count, and accuracy weight ω take the values reported in Section 5.4.

5 METHODOLOGY

This chapter describes how AFLOAT is evaluated in simulation, with each design's execution time, resource footprint, accuracy, and power taken from profiles measured on real FPGA hardware. All experiments run on a workstation with an AMD Ryzen 9 9900X processor and 32 GB of RAM, running Ubuntu 24.04. The chapter sets out the simulated hardware platform, the Design Pool that supplies its designs, the workloads that drive it, the configurations evaluated, the metrics collected, and the assumptions that bound the study.

5.1 Hardware Platform

The simulated multi-FPGA cluster is built from three FPGA devices covering three capacity tiers of a heterogeneous edge platform, from a small baseline device up to a much larger one. Table 5.1 lists the resources and static power of each.

Table 5.1 – FPGA devices of the simulated multi-FPGA cluster.

FPGA Device	LUTs	FFs	DSPs	BRAMs	Static Power
Artix-7 XC7A200T	134,600	269,200	740	365	0.122 W
Zynq-7000 XC7Z045	218,600	437,200	900	545	0.199 W
Kintex-7 XC7K410T	254,200	508,400	1,540	795	0.187 W

Source: the author.

The Artix-7 XC7A200T is the smallest device and sets the baseline resource budget. The Zynq-7000 XC7Z045 forms the mid tier, with roughly 1.6 times the LUTs of the Artix-7. The Kintex-7 XC7K410T is the largest, with about 1.9 times the LUTs and 2.1 times the DSPs of the Artix-7. The static power of each is taken directly from the Xilinx Power Estimator (XPE).

Because the three devices are not identical hardware, a design must be synthesized separately for each one it runs on. Its resource footprint, however, barely changes across these syntheses, so the evaluation characterizes each design once on the smallest Artix-7 and takes that footprint to hold across the cluster. The devices also form a strict capacity chain, the Zynq-7000 a resource superset of the Artix-7 across every resource type and the Kintex-7 a superset of the Zynq-7000. A design that fits the Artix-7 therefore fits the larger two as well, which differ only in holding more of its instances at once.

These three device types are the building blocks of the cluster, combined in different numbers to form platforms of increasing size. The experiments evaluate five configurations, scaling from 4 to 8 total devices, with the composition of each given in Table 5.2.

Table 5.2 – Cluster configurations used in the evaluation.

Configuration	Artix-7	Zynq-7000	Kintex-7	Total Devices
Tiny	2	1	1	4
Small	2	2	1	5
Medium	3	2	1	6
Large	3	2	2	7
Huge	3	3	2	8

Source: the author.

Scaling the cluster adds devices of specific tiers rather than growing all tiers uniformly. The Tiny configuration gives the most constrained resource budget, where provisioning decisions weigh most heavily on response time. The Huge configuration tests scalability with over twice the aggregate capacity of the Tiny. Every configuration includes all three device types, so the provisioning strategy must always reason about devices with different resource profiles.

Before a batch runs, its chosen designs must first be loaded onto the devices. This reconfiguration is a fixed setup cost that the simulation does not model.

5.2 Design Pool

The evaluation populates the two repositories AFLOAT draws on at *Static Time*, the Task Library and the Design Pool. The Task Library holds the task types the system serves, each a DNN inference problem defined by a model and a dataset, while the Design Pool holds the designs that run them, every design associated with one task type. This work uses four task types, one per image-classification dataset: MNIST (DENG, 2012) for handwritten-digit recognition, CIFAR-10 (KRIZHEVSKY; HINTON et al., 2009) for object classification, SOYBEAN (LIN et al., 2023) for soybean-seed classification, and GRAPE (MANDAL; GOUDA, 2024) for grapevine-disease classification. Each is a distinct inference pipeline, with its own model architecture, input dimensions, and classification difficulty. Together they give the framework a varied multi-tenant task set to provision over.

The designs were generated and characterized ahead of this evaluation, as Section 4.1.1 describes. The base models were taken pre-trained from the Brevitas repository for the standard benchmarks, CIFAR-10 and MNIST, and trained from scratch for the agricultural ones, SOYBEAN and GRAPE. Structured pruning was applied at six rates, 0%, 25%, 50%, 75%, 85%, and 90%, and mapping each resulting model variant to five FINN builds produced a Design Pool of 118 designs spanning wide ranges of accuracy, resource use, and throughput. Table 5.3 reports these characteristics per task type, each as the minimum and maximum across the type's designs. The utilization, the largest share any single resource type takes on the smallest device, bounds how many instances of a design a device can hold. CIFAR-10 contributes 28 designs rather than 30 because its two most aggressive pruning rates admit only four distinct builds each, the heavily reduced model leaving too few synthesizable folding configurations to fill all five throughput targets.

Table 5.3 – Design Pool characteristics by task type.

Metric	CIFAR-10	GRAPE	MNIST	SOYBEAN
Designs	28	30	30	30
LUT (k)	7.5–27.4	3.2–12.9	3.7–44.8	6.9–50.2
FF (k)	6.6–38.9	2.8–16.2	2.9–64.1	5.7–74.2
DSP	9–576	5–461	3–544	4–525
BRAM	16–234	2–49	0–16	6–87
Util. (%)	5.6–77.8	2.4–62.3	2.7–73.5	5.1–70.9
Dyn. Power (W)	0.11–0.79	0.04–0.42	0.04–0.67	0.08–1.04
Thpt (fps)	301–1.07k	339–78.1k	1.99k–5.56M	253–6.85k
Acc. (%)	51.4–88.6	75.6–98.1	62.2–96.6	84.9–96.4

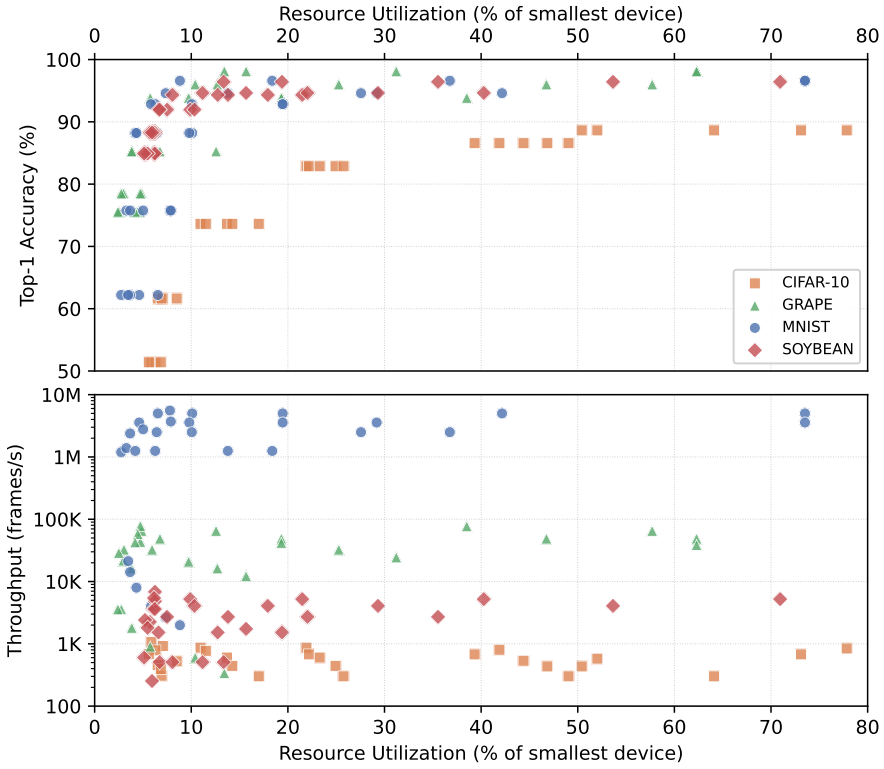
Source: the author.

Figure 5.1 plots each design against its resource footprint, with accuracy in Figure 5.1a and throughput in Figure 5.1b, so the two readings line up per design. The pool is diverse, and each task type occupies a distinct region. Accuracy rises steeply with footprint among the smallest designs and then plateaus, so most of the achievable accuracy comes well before the largest footprints. Throughput instead separates into bands by task type, the simple MNIST models reaching by far the highest frame rates and the heavier SOYBEAN and CIFAR-10 models the lowest.

5.3 Workload Generation

The experiments drive the framework with synthetic workloads that reproduce the varying demand of an edge node. Requests arrive as the Poisson process of Section 2.5,

Figure 5.1 – The Design Pool across accuracy, throughput, and resource footprint.
(a) Accuracy against resource footprint.



(b) Throughput against resource footprint.

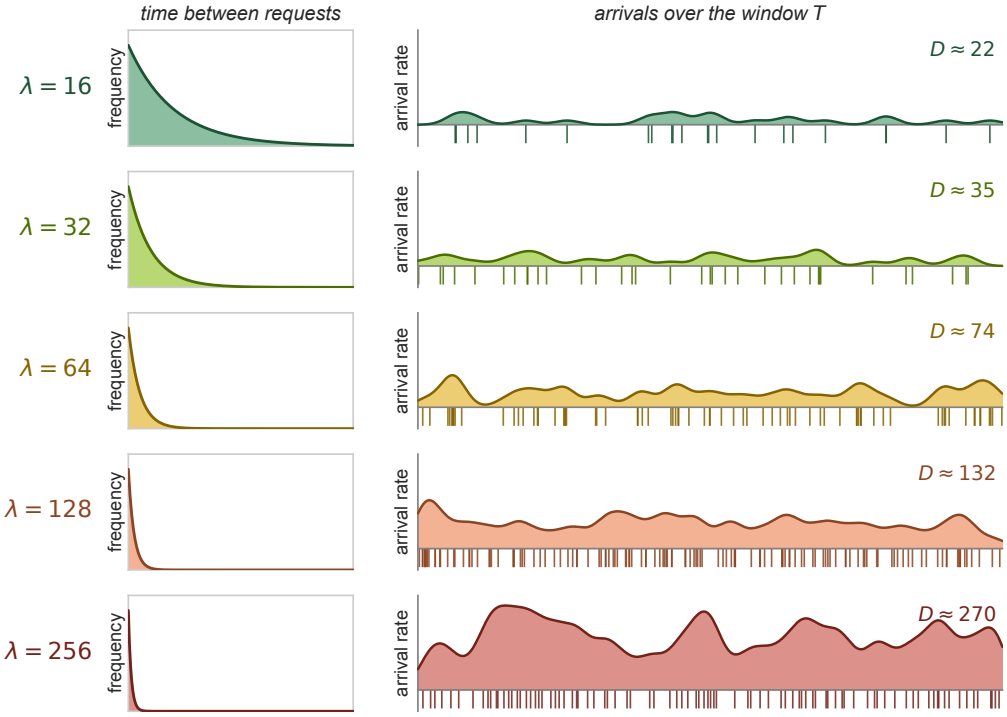
Source: the author.

gathered over a batch window held fixed at one second. With the window fixed, the demand each batch carries is set by the arrival rate, which the evaluation sweeps to vary the load. All four task types are active in every run, with each request drawn uniformly among them, so the framework always faces the full multi-tenant mix.

The arrival rate is set so that the demand gathered over the window roughly doubles across five levels: about 16, 32, 64, 128, and 256 requests. These span a lightly loaded cluster, where the most accurate designs can serve every request, up to a heavily overloaded one, where compact designs are needed for the parallelism to keep up. Figure 5.2 shows the arrivals each level produces over a single window, from the sparsest at the lightest level to the densest at the heaviest.

Requests also differ in volume: each carries a size factor, drawn uniformly from 1 to 4, that sets how many frames it processes. With a design's throughput fixed, this scales the request's execution time proportionally, so two otherwise identical requests can take very different times. The arrival rate that fixes each demand level accounts for the average request size, so the demand levels stay consistent from run to run regardless of the sizes drawn.

Figure 5.2 – Arrivals gathered over the window at the five evaluated demand levels.



Source: the author.

Every configuration is repeated over multiple independent runs, with the reported metrics averaging over the random request draws and sizes within each.

5.4 Evaluated Configurations

The evaluation sweeps the framework across four dimensions: the accuracy threshold, the provisioning strategy, the demand level, and the cluster configuration. Two fixed baselines mark the extremes it operates between, a calibration sets the adaptive window's contention parameters, and a separate ablation measures what each pipeline stage contributes.

The Static Accuracy Filter is evaluated at three thresholds, $\tau \in \{50\%, 75\%, 90\%\}$. No design in the pool falls below 50% accuracy, so that threshold removes nothing and serves as the unfiltered reference. The 75% threshold drops the least accurate designs while keeping many compact ones, and 90% leaves only the most accurate few.

Fourteen strategy configurations run the full pipeline, each pairing the Static Accuracy Filter and the Adaptive Contention-Aware Window with a provisioning strategy. The window uses the contention sensitivity and window fraction selected by the calibration described in Section 5.4.1, and pins the single highest-accuracy design of each task type

so a high-quality option is always reachable. The strategies sweep the accuracy weight ω:

- **Genetic algorithm** at $\omega \in \{0,\ 1,\ 2,\ 4,\ 8,\ 16,\ 32\}$, seven configurations from pure response-time optimization at $\omega = 0$ to a strong accuracy preference at $\omega = 32$.
- **Greedy heuristic** at $\omega \in \{0,\ 1,\ 2,\ 4,\ 8,\ 16,\ 32\}$, seven configurations over the same range.

Raising ω traces the accuracy–response-time trade-off, steering each strategy from minimizing response time alone toward favoring more accurate designs. The GA uses a population of $P = 200$ evolved over at most $G = 500$ generations, with tournament size 3, one-point crossover on every selected pair, a mutation rate $\mu = 0.2$, the best individual preserved each generation, and an early exit after $p = 15$ generations without improvement. Crossing the three thresholds, five demand levels, five cluster configurations, and fourteen strategies, each repeated over 50 runs, gives 52,500 simulations.

Two fixed baselines bound the accuracy–response-time spectrum the framework works within, bypassing both filtering stages. The Highest-Accuracy baseline keeps only the single most accurate design of each task type, fixing the upper bound on accuracy and, with it, on response time. The Highest-Throughput baseline keeps only the single fastest design, fixing the lower bound on both. Each baseline is provisioned by the GA and Greedy at $\omega = 0$, over the same five demand levels and five cluster configurations at 50 runs each, adding 5,000 simulations.

5.4.1 Contention-Parameter Calibration

The contention parameters of the Adaptive Contention-Aware Window, the sensitivity s and the window fraction f_w, are calibrated separately rather than varied in the main evaluation. This calibration runs on the Medium cluster across the three thresholds, varying s from 0.01 to 0.05 and f_w from 0.35 to 0.65 together with the accuracy weight $\omega \in \{0,\ 1,\ 2,\ 4,\ 8\}$. Across the five demand levels at 40 runs each, the calibration adds 105,000 simulations. The s and f_w it selects are fixed for the rest of the evaluation and reported in Section 6.2.

5.4.2 Contribution of the Pipeline Stages

The ablation runs on one fixed setup, the Small cluster with the GA at $\omega = 1$, sweeping the five demand levels, and toggles the two pipeline stages independently:

- **Unrestricted:** neither stage, the strategy works on the full Design Pool.
- **Filter only:** the accuracy filter at $\tau = 75\%$, without the window.
- **Window only:** the adaptive window, without the accuracy filter.
- **Full pipeline:** the filter at $\tau = 75\%$ followed by the window.

Where the window is active, the ablation uses a more aggressive window than the calibration selects (Section 6.2), with the sensitivity raised to $s = 0.15$ and the fraction narrowed to $f_w = 0.30$. This sharpens the window's response to rising demand, widening the separation between the configurations that include it and those that do not.

Across the five demand levels at 50 runs each, the ablation adds 1,000 simulations.

5.5 Metrics

The experiments record several metrics. The primary pair is the response time and accuracy that the framework trades against each other. The rest characterize each allocation and the decision behind it: its makespan, energy, device occupancy, and the time the provisioning strategy takes to converge.

The main timing metric is the Response Time, $\overline{RT}$, the average time each task takes from the moment its batch is complete until its inference finishes. A batch is complete at its last arrival t_{last}, after which provisioning begins, so the response time is measured from that point:

$$\overline{RT} = \frac{1}{N} \sum_{i=1}^{N} (t_{\text{finish},i} - t_{\text{last}}) \tag{5.1}$$

where N is the number of tasks in the batch and $t_{\text{finish},i}$ is task i's completion time. Traditionally a task's response time runs from its own arrival. Measuring instead from the last arrival isolates the latency the framework controls, the provisioning decision and the resulting contention, from the wait while the batch gathers. The two definitions differ only by that gather wait, whose mean over the batch is

$$\overline{g} = \frac{1}{N} \sum_{i=1}^{N} (t_{\text{last}} - t_{\text{arrival},i}), \tag{5.2}$$

so the conventional from-arrival response time is just $\overline{RT} + \overline{g}$. Since $\overline{g}$ depends only on the batch window T (Section 5.3) and the arrival times, not the allocation, it shifts every configuration by the same amount, bounded by T and about $T/2$ on average, and leaves their comparison unchanged.

Accuracy is reported as the Average Top-1 Accuracy, the accuracy of each task's design averaged over the batch:

$$\overline{Acc} = \frac{1}{N} \sum_{i=1}^{N} \text{Acc}(d_i) \tag{5.3}$$

where $\text{Acc}(d_i)$ is the Top-1 accuracy of the design d_i assigned to task i. This is the inference quality the framework trades against response time.

Beyond the response time and accuracy it trades, each allocation is characterized by several further quantities, reported but not optimized:

- **Makespan:** the total time to finish the whole batch, from the first task starting to the last completing. Where the response time is a per-task average that dilutes a slow minority, the makespan is set by that minority, exposing the tail the mean hides.
- **Energy:** the total energy the active devices consume over the batch, accumulating each one's static and dynamic power over the time it runs.
- **Convergence Time:** the time the provisioning strategy spends searching before its allocation can run, an overhead paid once per batch that the response time does not include.
- **Device Occupancy:** the number of design instances each device hosts, showing how the allocation spreads work across the heterogeneous cluster. A design's resource footprint bounds how many instances a device can hold.

Beyond the aggregate metrics, the allocations are studied directly, through which designs each strategy deploys from the pool and how that selection shifts as demand varies.

5.6 Assumptions and Limitations

This work rests on a number of assumptions about the framework and accepts some limitations in how it is evaluated, collected here with their practical implications.

Simulation-based evaluation. The provisioning and execution of each batch are simulated rather than run on physical FPGAs, though each design's resource use, timing, accuracy, and power are taken from real synthesis and characterization (Section 5.2). A real deployment would also involve reconfiguring the devices between batches and dispatching tasks over a communication fabric, which a future version of the simulator could model.

Synthetic workloads. The workloads are synthetic, generated from the Poisson process rather than captured from real edge traffic. They are meant to probe the framework's response across demand levels, not to reproduce a particular deployment.

Resource fitting. A design is taken to fit a device when its LUT, FF, DSP, and BRAM demands are within capacity. Each design places, routes, and closes timing on its own, but this is not verified when several share a device.

Multi-tenant bitstreams. Co-locating several FINN accelerators on one bitstream has been prototyped and validated, so the approach is feasible. The bitstreams are prepared ahead of provisioning, since synthesis is not feasible at *Dynamic Time*. The number of multi-accelerator combinations grows combinatorially with the pool, which becomes impractical to deploy for a large one. These deployment challenges are left to future work, where viable solutions can be found.

6 RESULTS

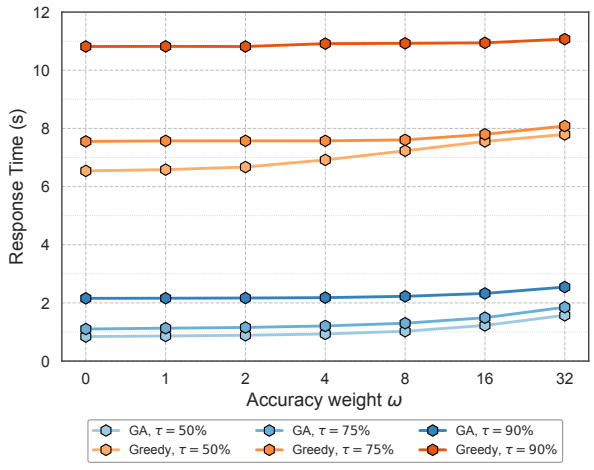
This chapter presents the results of evaluating AFLOAT on the simulated cluster. It first compares the two provisioning strategies, then calibrates the adaptive window and establishes the range the framework operates within. It closes by examining the effect of demand, the scaling across platform sizes, and the contribution of each pipeline stage.

6.1 Provisioning Strategy Quality

AFLOAT implements two provisioning strategies that search the allocation space in opposite ways (Section 4.2.3): a greedy heuristic (Greedy) that commits to a single allocation in one deterministic pass, and a genetic algorithm (GA) that evolves a population of candidates before settling on one. This section compares the two across the three accuracy thresholds τ and the accuracy weight ω, characterizing how each balances response time, the per-task average from batch gathering to inference finish (Section 5.5), against accuracy. To isolate the effect of the strategy and its settings, every value averages over all demand levels and cluster sizes, examined separately in Sections 6.4 and 6.5.

Figure 6.1 plots response time on the vertical axis against the accuracy weight ω on the horizontal one. A higher ω steers each strategy toward more accurate, heavier designs (Section 4.2.3). Color identifies the strategy, blue for the GA and orange for Greedy, while shade marks the threshold τ, lighter for looser and darker for stricter ones. A higher τ raises the floor the static filter enforces, leaving fewer designs eligible (Section 4.2.1).

Figure 6.1 – GA and Greedy response time across the accuracy weight ω, by threshold τ.

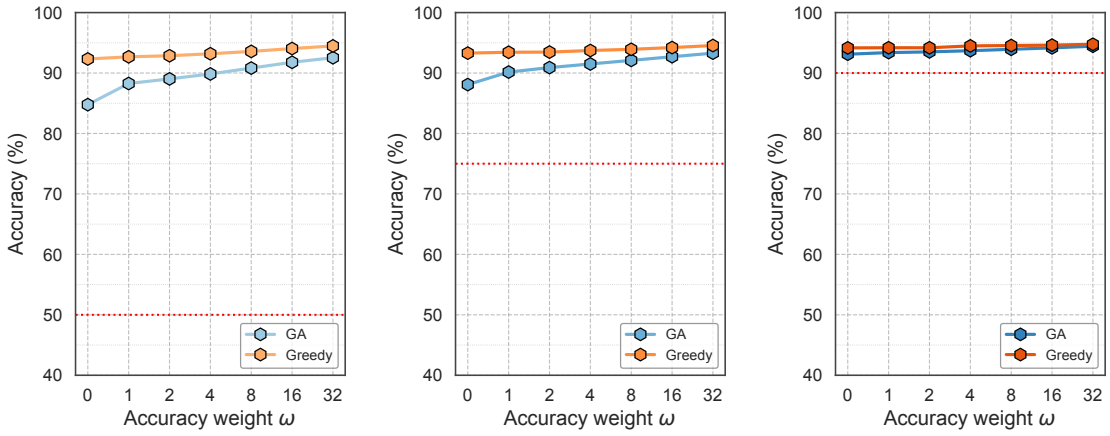


Source: the author.

The two strategies occupy clearly separated bands: the GA holds below three seconds across the entire sweep, while Greedy never drops below six. This separation reflects how each searches the Active Pool, Greedy committing to a single pass that packs the boards loosely while the GA evaluates whole allocations and finds the denser packings that bring response time down. Within both bands the threshold stacks the lines the same way, response time rising with τ as a stricter floor admits only heavier, more accurate designs. The looser thresholds respond most to ω, climbing as the weight pulls selection toward heavier designs, while the strict $\tau = 90\%$ lines stay nearly flat, their pool already too narrow for the weight to shift.

Figure 6.2 reports the accuracy behind these response times, keeping the same horizontal axis and strategy colors but splitting the thresholds into separate panels on a shared accuracy scale. The dotted line in each panel marks its threshold τ, the accuracy floor the static filter enforces (Section 4.2.1).

Figure 6.2 – GA and Greedy accuracy across the weight ω, one panel per threshold τ.
(a) $\tau = 50\%$ (b) $\tau = 75\%$ (c) $\tau = 90\%$



Source: the author.

In every panel both strategies stay above their threshold and climb toward the same ceiling as ω rises. The GA trails Greedy at low ω, by several percentage points (pp) at the loosest threshold, because its search leans on response time while accuracy carries little weight. The threshold acts as a hard floor here, the strict $\tau = 90\%$ keeping both high while the loose $\tau = 50\%$ lets the GA drop much lower as it trades accuracy for speed. Raising ω therefore buys back several percentage points of accuracy for only a slight rise in response time, letting the GA approach Greedy's accuracy while holding on to its large response-time advantage.

That response-time advantage comes at the cost of a longer search, reported as convergence time in Table 6.1. Greedy converges in a near-constant time that hardly

moves across thresholds, its single pass doing the same work however many designs pass the filter, and it is equally insensitive to demand and cluster size. The GA's search costs more, but that cost falls sharply as a tighter threshold leaves fewer designs to search. Demand pushes it the other way, with each added task increasing the work of scoring every candidate, while a larger cluster changes it little, since more boards add capacity rather than deepening the search, as Tables 6.3 and 6.4 later show. Even at its highest, this overhead remains far smaller than the seconds the GA saves in response time.

Table 6.1 – Convergence time of the GA and Greedy strategies.

Strategy	$\tau = 50\%$	$\tau = 75\%$	$\tau = 90\%$
GA	99ms	85ms	34ms
Greedy	3ms	3ms	3ms

Source: the author.

Beyond reaching far lower response times, the GA varies far more with τ and ω, spanning much of the accuracy–response-time trade-off where Greedy stays comparatively fixed. This range reveals the framework's behavior more clearly, so the rest of this chapter focuses on the GA and keeps Greedy as a reference point.

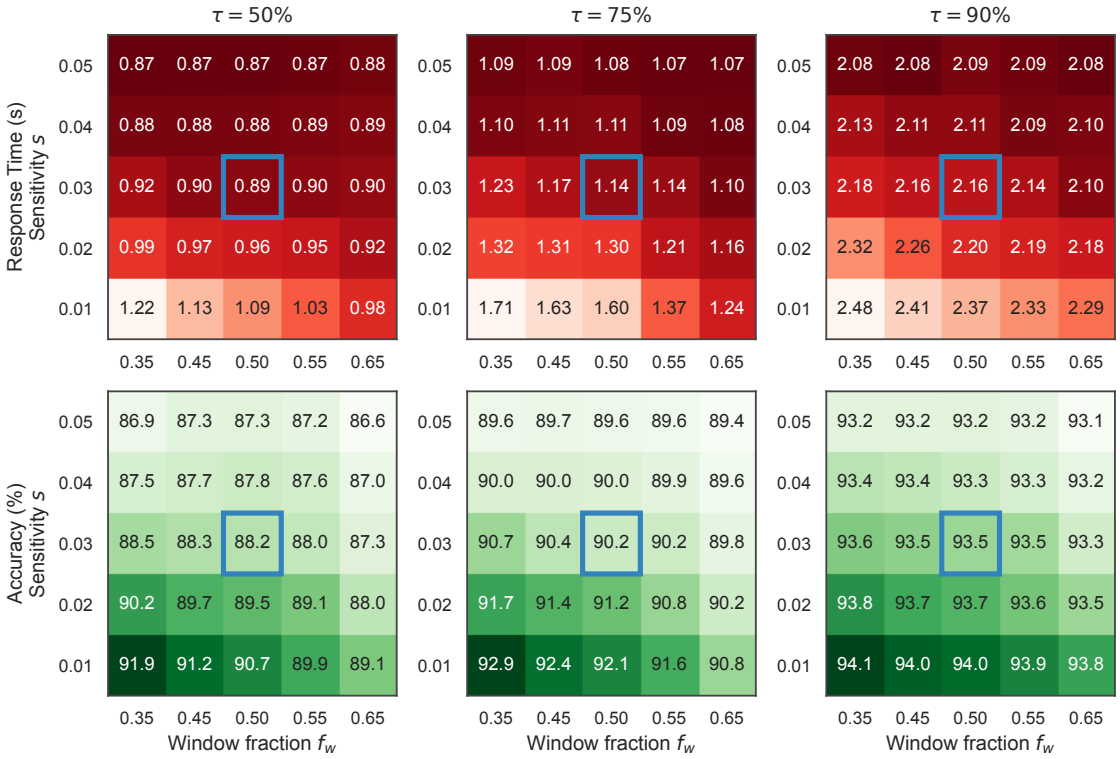
6.2 Calibrating the Adaptive Window

The adaptive window is governed by two contention parameters, the sensitivity s and the window fraction f_w (Section 4.2.2). This section calibrates them, sweeping s and f_w over a grid and recording the response time and accuracy each combination produces (Section 5.4.1).

Figure 6.3 shows the sweep as a grid of heatmaps. The top row holds response time and the bottom accuracy, one column per threshold τ. Within each, the sensitivity s sits on the vertical axis and the window fraction f_w on the horizontal. A higher s slides the window toward compact designs sooner, and a wider f_w admits more designs at once.

Both parameters move the system along the same trade-off. Raising s pulls response time down toward a floor while giving up accuracy. This cost shrinks as the threshold tightens, nearly vanishing at $\tau = 90\%$. Response time reaches its floor by $s = 0.03$, beyond which a higher sensitivity adds almost no speed but keeps lowering accuracy. The window fraction f_w has a gentler effect, a wider window costing accuracy while response time stays nearly flat, so a moderate 0.50 avoids that loss. The boxed $s = 0.03$, $f_w = 0.50$ configures the adaptive window through the rest of the chapter.

Figure 6.3 – Response time and accuracy across the sensitivity s and window fraction f_w, per threshold τ. The chosen $s = 0.03$, $f_w = 0.50$ is boxed.



Source: the author.

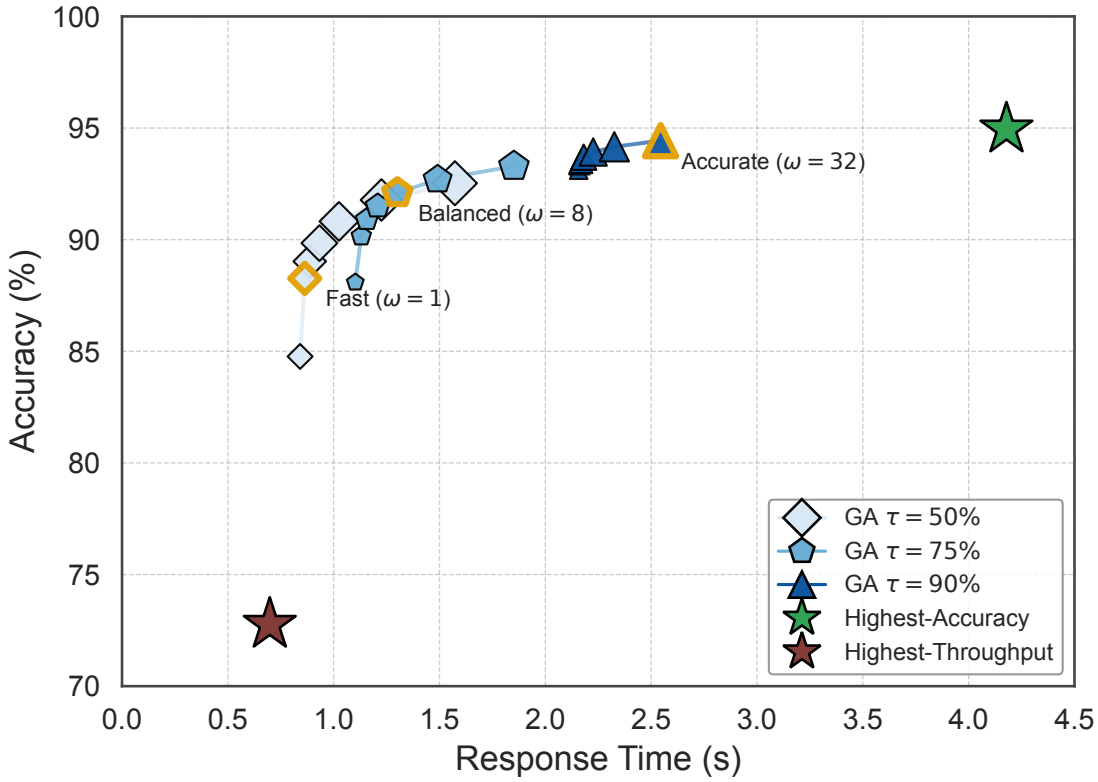
6.3 Baselines and Featured Configurations

The rest of the chapter follows a small set of configurations, established here. Two baselines mark the extremes the framework works between (Section 5.4). The Highest-Accuracy baseline deploys only the most accurate design of each task type and Highest-Throughput only the fastest, so neither adapts to load.

Figure 6.4 places the GA configurations in the accuracy–response-time plane. Response time runs along the horizontal axis and accuracy up the vertical. Each threshold τ has its own color and marker, tracing a frontier whose points grow with ω. The two baselines sit as stars at opposite corners, and the three chosen configurations are outlined in gold.

The points climb from the fast, low-accuracy corner near Highest-Throughput toward the slow, high-accuracy corner near Highest-Accuracy. Tightening τ shifts the whole frontier up and to the right, onto a higher-accuracy, slower band. Raising ω instead moves gradually along a single frontier. That range is wide at the looser thresholds but shrinks to almost nothing at $\tau = 90\%$, where the narrow pool leaves the window little to move through.

Figure 6.4 – GA frontiers per threshold τ between the two baselines. The featured configurations are outlined in gold.



Source: the author.

Table 6.2 lists these configurations with their response time and accuracy. The three GA configurations are Fast at $(\tau, \omega) = (50\%, 1)$, Balanced at $(75\%, 8)$, and Accurate at $(90\%, 32)$. Fast lands close to Highest-Throughput, within **24%** of its response time while its accuracy stays more than **15 pp** higher. Balanced sits between, cutting response time by **69%** compared with Highest-Accuracy while staying **19 pp** more accurate than Highest-Throughput. Accurate sits at the other end, matching Highest-Accuracy to within **0.5 pp** at **40%** less response time.

Table 6.2 – Featured GA configurations and the two baselines.

Configuration	Provisioner	τ	ω	Response Time	Accuracy
Highest-Accuracy	GA	–	–	4.2s	94.9%
Accurate	GA	90%	32	2.5s	94.4%
Balanced	GA	75%	8	1.3s	92.1%
Fast	GA	50%	1	863ms	88.3%
Highest-Throughput	GA	–	–	698ms	72.8%

Source: the author.

These five configurations appear in all the results that follow.

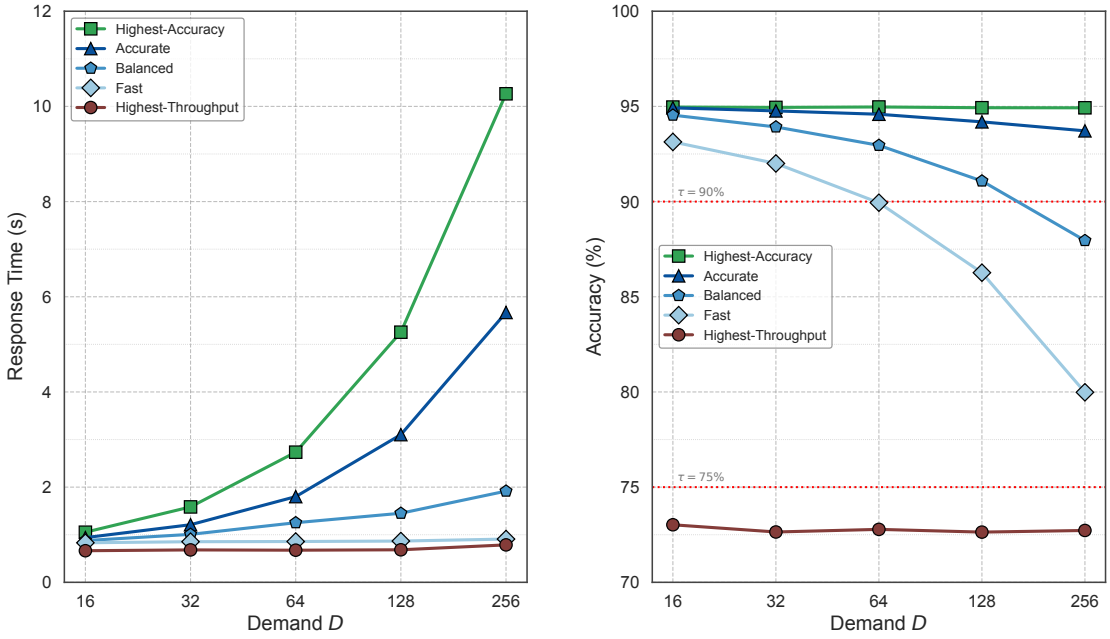
6.4 Effect of Demand

AFLOAT is built to navigate the accuracy–response-time trade-off as demand varies (Chapter 4). This section raises the demand D across its five levels (Section 5.3) and follows how each configuration responds.

Figure 6.5 plots each configuration as a line against demand D on the horizontal axis. The left panel shows response time and the right average accuracy. Dotted lines in the accuracy panel mark the thresholds τ enforced on the configurations.

Figure 6.5 – Response time and accuracy across demand D for the featured configurations and the two baselines.

(a) Response time. (b) Average accuracy.

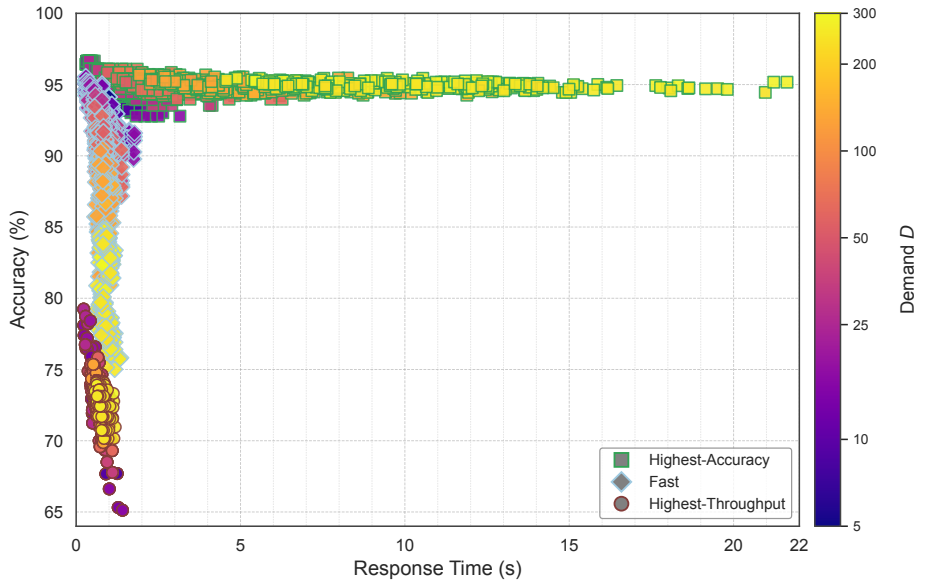


Source: the author.

At low demand the configurations sit close together, then fan out between the two baselines as load grows. Highest-Accuracy and Accurate hold their accuracy and pay in response time, the former climbing past **10 s** at peak load. Fast goes the other way, holding response time nearly flat while accuracy falls from **93%** to **80%**. Each adaptive configuration trades accuracy for parallelism as demand rises, bounded by its threshold.

Figure 6.6 returns to the accuracy–response-time plane of Figure 6.4, now only for the Fast configuration and the two baselines. Each run appears as one point, its response time on the horizontal axis and accuracy on the vertical. Marker shape and edge color identify the configuration, while the fill encodes the run's demand D on the color bar, dark at low demand and bright at high.

Figure 6.6 – Per-run response time and accuracy of the Fast configuration and the two baselines, colored by demand D.



Source: the author.

Fast forms a near-vertical column, its response time anchored near the floor while accuracy slides down as demand grows. The lightest-demand runs sit at the top and the heaviest at the bottom. Highest-Accuracy spreads the opposite way, a flat band pinned at high accuracy whose response time stretches far to the right as load grows. Highest-Throughput stays a small cluster at the low-accuracy floor. Each configuration forms a distinct cloud.

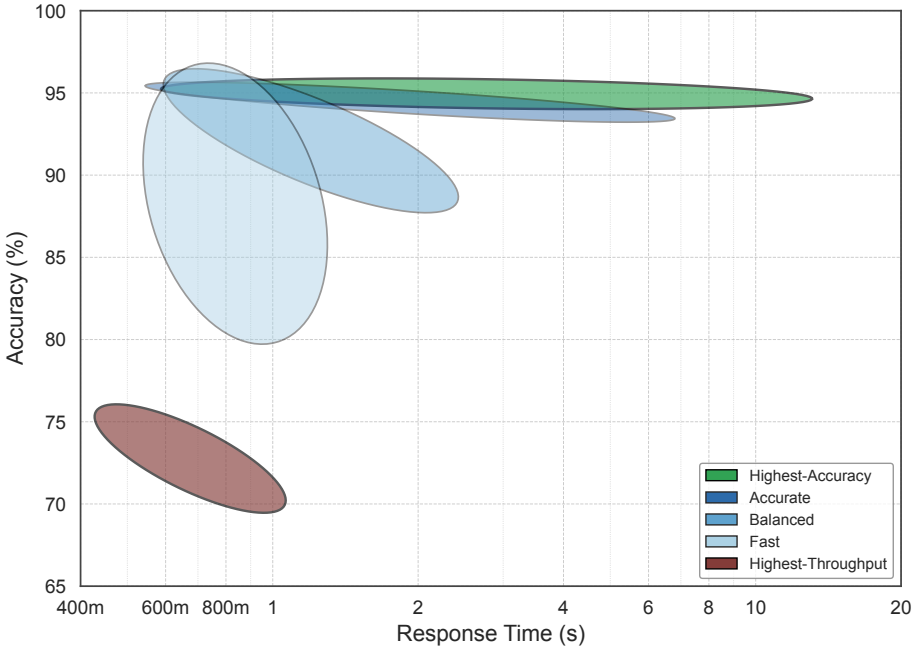
Figure 6.7 brings all five configurations together, drawing each cloud as a 75% confidence ellipse in its own color. The response-time axis is logarithmic, since it ranges widely across the configurations.

Each configuration operates in its own region of the plane. Within each region, the lightest-load runs sit at one end and the heaviest at the other. Fast forms the roundest, its accuracy swinging while response time holds low. Balanced and Accurate lean progressively flatter, their accuracy spread narrowing and their response-time spread widening as the threshold tightens. The two baselines bound the plane, the GA regions leaning between them.

Figure 6.8 shows which designs execute each configuration's tasks. Each bar splits a configuration's executed tasks among the pruning levels, the bars grouped by demand D. Color marks the pruning level and hatching the configuration.

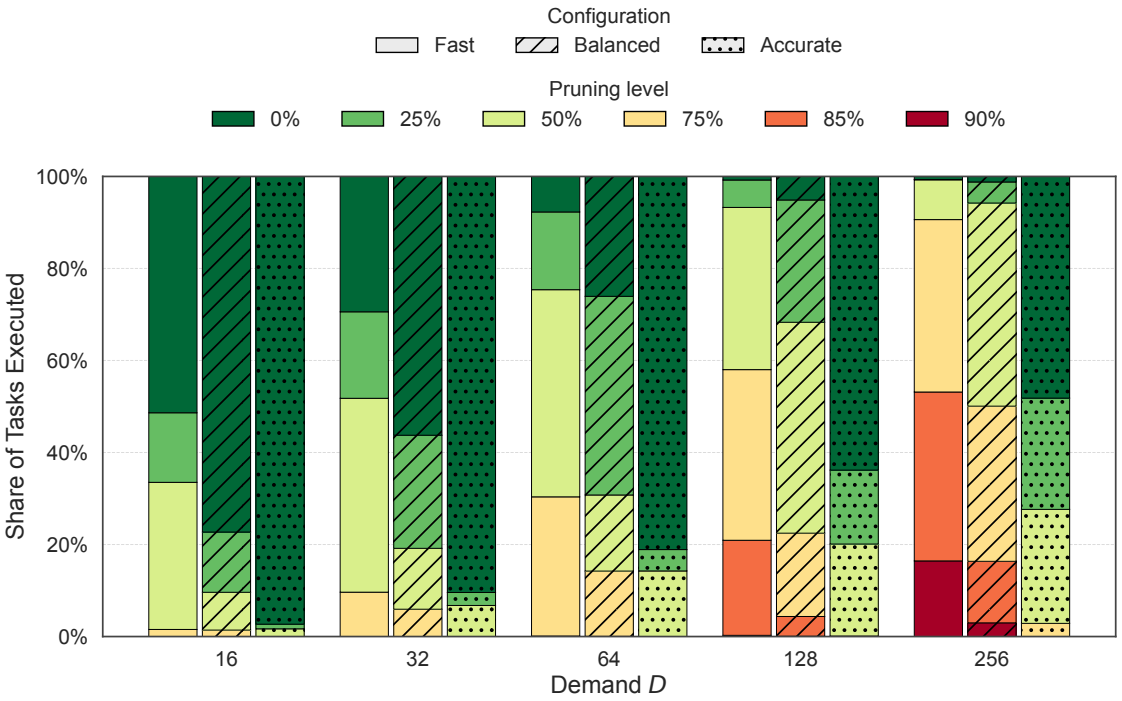
Fast's design mix shifts the most across demand. At low demand it leans on lightly pruned designs, sliding onto the most compact ones as load climbs. Accurate shifts far less,

Figure 6.7 – Confidence regions of the featured configurations and the two baselines, each a 75% ellipse.



Source: the author.

Figure 6.8 – Pruning composition of the deployed designs per configuration, by hatching, across demand D.



Source: the author.

never reaching the compact designs, since its strict threshold filtered them out. Balanced sits between the two. The static filter limits how far each window can slide, which shapes each configuration's region.

Table 6.3 reports the secondary costs at each demand level: the makespan (the time to finish the whole batch), energy, and convergence time.

Table 6.3 – Makespan, energy, and convergence time across demand levels.

Configuration	Makespan (s)					Energy (J)					Conv. Time (ms)				
	16	32	64	128	256	16	32	64	128	256	16	32	64	128	256
Highest-Accuracy	4.6	8.1	14.9	30.0	59.0	13	27	51	102	208	2	3	6	12	26
Accurate	4.1	6.1	10.3	19.3	35.8	13	27	50	98	198	3	6	12	28	95
Balanced	3.9	5.1	7.1	8.7	12.0	12	22	37	54	88	4	8	20	56	207
Fast	3.7	4.2	4.4	4.7	5.3	7	12	18	27	36	5	10	27	92	362
Highest-Throughput	3.0	3.4	3.6	3.7	4.6	2	5	8	13	25	1	2	3	11	98

Source: the author.

Makespan and energy climb with demand for every configuration, steepest for Accurate and Highest-Accuracy, whose long-running designs stretch the batch out. Fast and Highest-Throughput stay cheap on both. Convergence time climbs steeply with demand, worst for Fast, whose loose threshold leaves the most designs to search. The Greedy heuristic would avoid this growth with its near-instant single pass (Section 6.1), but its weaker allocations would raise makespan and energy instead.

6.5 Heterogeneous Platform Scaling

A deployment may run on a cluster of any size. This section holds demand at its peak and steps through five heterogeneous clusters, from Tiny to Huge (Section 5.1).

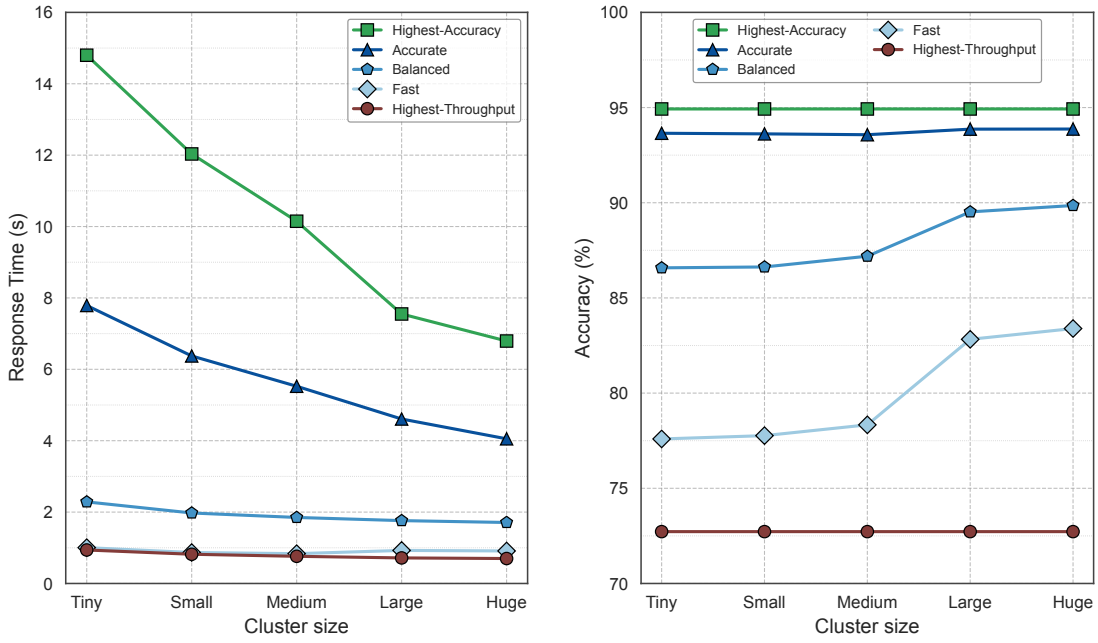
Figure 6.9 plots each configuration as a line against cluster size on the horizontal axis. The left panel shows response time and the right average accuracy.

More boards mainly lower response time. Highest-Accuracy and Accurate shed seconds as the extra parallelism clears their heavy designs faster, while Fast and Highest-Throughput, already near the floor, fall only slightly. In accuracy, the two baselines stay flat, and Accurate with them, already near the top. Fast and Balanced instead climb with cluster size, with a clear jump at Large. The larger capacity drops the load factor ρ, so their window holds higher-accuracy designs. What the baselines can only turn into speed, the adaptive configurations also turn into accuracy.

Figure 6.10 gives the instances each configuration hosts per board, across cluster sizes.

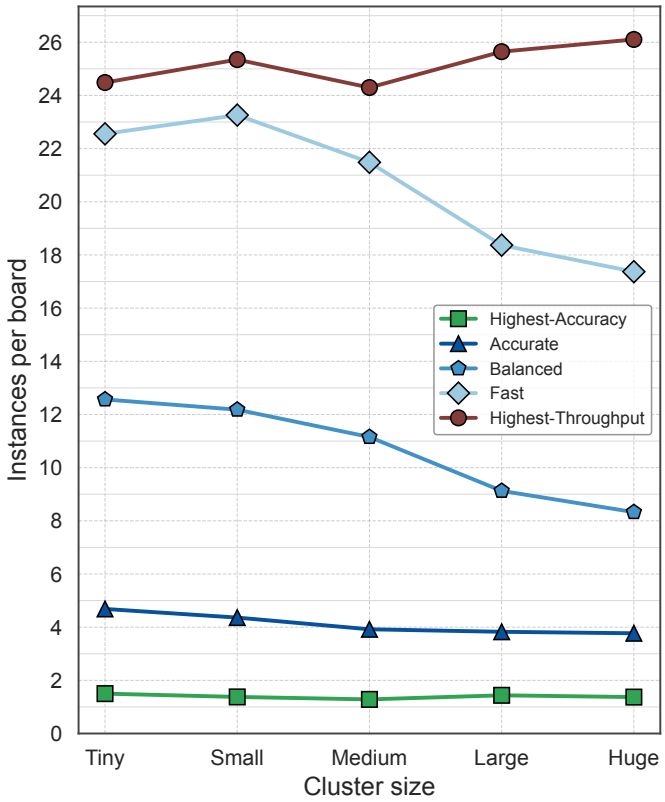
Figure 6.9 – Response time and accuracy across cluster size for the featured configurations and the two baselines, at peak demand.

(a) Response time. (b) Average accuracy.



Source: the author.

Figure 6.10 – Instances per board per configuration across cluster sizes, at peak demand.



Source: the author.

Highest-Throughput packs the most instances per board, Highest-Accuracy the fewest, and the rest fall between. As the cluster grows, the adaptive configurations ease their per-board count, the freed space going to heavier, more accurate designs. This is the same relief that lowers response time and lifts accuracy.

Table 6.4 gives the secondary costs across the cluster sizes at peak demand, each abbreviated to its initial.

Table 6.4 – Makespan, energy, and convergence time across cluster sizes at peak demand.

Configuration	Makespan (s)					Energy (J)					Conv. Time (ms)				
	T	S	M	L	H	T	S	M	L	H	T	S	M	L	H
Highest-Accuracy	89.1	67.1	57.5	42.9	38.3	208	210	208	206	207	20	21	22	31	34
Accurate	49.6	40.7	34.9	28.7	25.2	197	199	198	198	200	78	87	86	107	117
Balanced	14.3	12.4	11.7	10.9	10.7	82	84	86	92	95	172	205	205	222	231
Fast	5.9	5.0	4.8	5.5	5.3	29	30	31	45	47	316	360	375	376	385
Highest-Throughput	5.7	5.0	4.4	4.0	3.8	24	24	24	25	25	90	106	114	102	80

Source: the author.

Makespan falls as the boards parallelize the batch, steepest for Accurate and Highest-Accuracy. Energy holds near-flat for Highest-Accuracy, which does the same work on any cluster, but climbs for Fast, whose relieved window deploys heavier designs that draw more power. Convergence time rises only gently here. The search depends far more on the number of tasks than the number of boards, so a larger cluster costs it much less than heavier demand did.

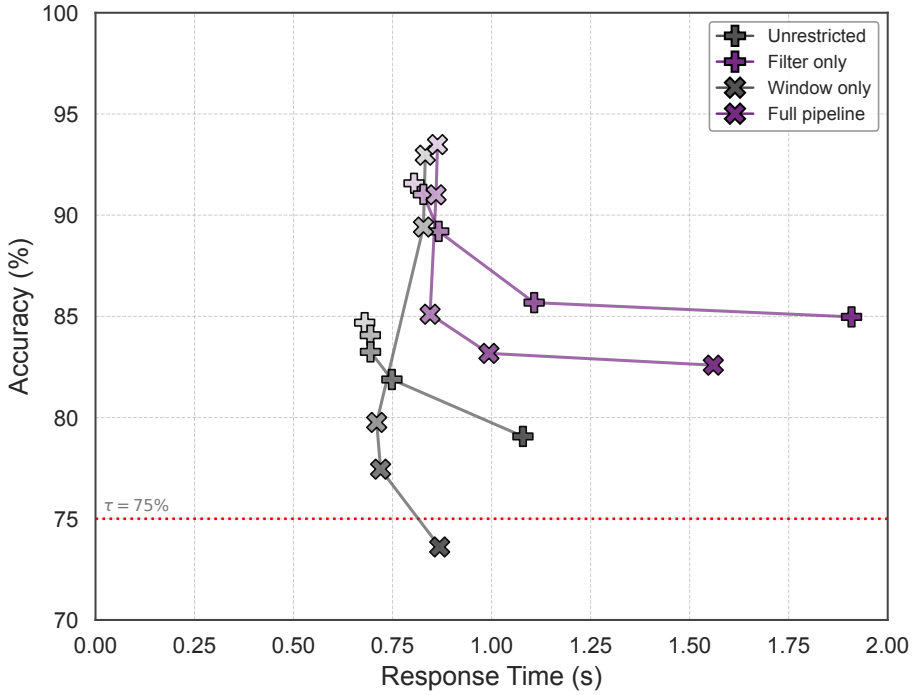
6.6 Contribution of the Pipeline Stages

The static filter and the adaptive window were built to address orthogonal concerns (Section 4.2.2). Toggling each stage on and off gives four pipeline configurations, evaluated on the Small cluster with the GA at $\omega = 1$ (Section 5.4).

Figure 6.11 places each configuration in the accuracy–response-time plane, its mean point at each demand joined into a path. Color marks the static filter, purple for on and gray for off. The marker shows the adaptive window, a cross for on and a plus for off. Shade tracks demand from light to dark, and the dotted line is the $\tau = 75\%$ floor.

With neither stage, the Unrestricted configuration stays in a low accuracy band throughout, at low response time. Each single-stage configuration holds either accuracy or response time, but not both. Filter only keeps accuracy high, but with no window to shift focus toward compact designs its response time climbs under load. Window only shifts toward compact designs to hold response time down, but with no floor its accuracy

Figure 6.11 – Mean accuracy and response time of each pipeline configuration across demand D, markers darkening from $D = 16$ to 256.



Source: the author.

drops below the $\tau = 75\%$ line.

The Full pipeline has both, and its path sits between the two. It shifts toward compact designs like Window only, holding response time below Filter only under load, while the filter keeps accuracy above the floor. Each stage covers the other's failure: the filter supplies the floor, the window the adaptation.

7 CONCLUSION

This work developed AFLOAT, a provisioning framework for DNN inference on heterogeneous multi-FPGA edge systems, where the choice of which accelerator designs to instantiate governs both the response time and the accuracy the cluster delivers. Because varying demand keeps that choice from settling on any fixed point, AFLOAT navigates the accuracy–response-time trade-off through three stages: a static accuracy filter that sets a quality floor over the Design Pool, an adaptive contention-aware window that shifts the eligible designs from high-accuracy toward compact ones as contention rises, and a provisioning strategy that allocates designs to boards and distributes tasks to instances (Chapter 4).

The framework was evaluated in simulation on heterogeneous clusters of four to eight FPGAs across five demand levels (Chapter 5). Of its two provisioning strategies, the genetic algorithm reached far lower response times than the greedy heuristic at a small cost in decision time (Section 6.1).

The framework's configurations spread between the two baselines, three of them featured as Fast, Balanced, and Accurate. Fast held response time within **24%** of Highest-Throughput while its accuracy stayed more than **15 percentage points** (pp) higher. Balanced cut response time by **69%** compared with Highest-Accuracy while staying **19 pp** more accurate than Highest-Throughput. Accurate matched Highest-Accuracy to within **0.5 pp** of accuracy at **40%** less response time (Section 6.3). As demand climbed, the adaptive configurations traded accuracy gradually to hold response time in check, an adaptation neither baseline could make (Section 6.4). Larger clusters turned into accuracy as well as speed (Section 6.5), and isolating the pipeline confirmed the two stages are complementary (Section 6.6).

The evaluation is simulation-based. Each design's characteristics come from real synthesis, but the reconfiguration and dispatch costs of a physical deployment are left out. The workloads are synthetic rather than captured from real edge traffic. The designs are considered pre-synthesized, and while co-locating them on one bitstream is feasible, the combinations to prepare grow combinatorially with the pool (Section 5.6).

7.1 Future Work

These limitations and the framework's modular design point to several directions for further work.

- **Physical deployment.** Running AFLOAT on real multi-FPGA hardware would validate the simulated trends and measure the reconfiguration and dispatch costs the simulation leaves out.
- **Partial reconfiguration.** Reconfiguring one region of a device at a time would let accelerators be loaded and swapped at runtime, without rebuilding its whole bitstream. The framework could then assemble the needed combinations on demand rather than preparing every one in advance.
- **Closed-loop adaptation.** Driving the adaptive window from contention measured at runtime would let it react to load rather than estimate it from demand alone.
- **Power-aware provisioning.** Powering down boards left idle under light demand would extend the framework from the accuracy–response-time trade-off to energy.
- **Per-task accuracy thresholds.** A separate accuracy threshold per task could replace the single global one, matching each task's own quality requirement.
- **Additional provisioning strategies.** The modular third stage accepts any allocation method, so strategies beyond the genetic algorithm and greedy heuristic, such as other metaheuristics, could be explored and compared.

7.2 Publications

The research behind this work has resulted in the following peer-reviewed publications:

- ADARE: Adaptive Resource Provisioning in Multi-FPGA Edge Environments (KERSZ et al., 2024).
- Exploiting Design Flexibility in Multi-Tenant Multi-FPGA Edge Systems (KERSZ et al., 2025c).
- HEDGY: Heterogeneous Design Management for Multi-Tenant Multi-FPGA Edge Systems (KERSZ et al., 2025b).
- Exploring Adaptive Design Mapping and Device Activation in Multi-FPGA Edge

Environments (KERSZ et al., 2025a).

- HARA: Hardware-Aware Resource Allocation for FPGA-Based Neural Network Acceleration (ELY et al., 2025b).
- NEAT: A Scalable SoC Framework for Adaptive Neural Network Acceleration in Edge Computing (ELY et al., 2025a).

INSTITUTIONAL ACKNOWLEDGEMENTS

This work has been partially funded by the project "Venture Science - Programa de Bolsas de Formação", supported by the Center for Embedded Devices and Research in Digital Agriculture (CEDRA) - SENAI-RS, with financial resources from the PPI IoT/Manufatura 4.0 / PPI HardwareBR of the MCTI, grant number 056/2023, signed with EMBRAPII, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CNPq, FAPERGS Techfuturo - 23/2551-0002199-4, the Office of Naval Research Global (ONR) (N629092212014), and PADIS through HT Micron Semicondutores.

REFERENCES

BHATT, P. D.; SHARMA, S. Analyzing impact of different inter-arrival time distributions on the performance of iot systems. In: **2022 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)**. [S.l.: s.n.], 2022. p. 243–248.

BIOOKAGHAZADEH, S.; RAVI, P. K.; ZHAO, M. Toward multi-fpga acceleration of the neural networks. **ACM JETC**, ACM New York, NY, USA, v. 17, n. 2, p. 1–23, 2021.

BIOOKAGHAZADEH, S.; RAVI, P. K.; ZHAO, M. Toward multi-fpga acceleration of the neural networks. **J. Emerg. Technol. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 17, n. 2, abr. 2021. ISSN 1550-4832.

BLOTT, M. et al. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. **ACM Transactions on Reconfigurable Technology and Systems (TRETS)**, ACM New York, NY, USA, v. 11, n. 3, p. 1–23, 2018.

CHEN, J. et al. Rich: Heterogeneous computing for real-time intelligent control. In: **2025 Design, Automation & Test in Europe Conference (DATE)**. [S.l.: s.n.], 2025. p. 1–7. ISSN 1558-1101.

CHENG, Y. et al. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. **IEEE Signal Processing Magazine**, v. 35, n. 1, p. 126–136, Jan 2018. ISSN 1558-0792.

CORMEN, T. H. et al. **Introduction to algorithms**. [S.l.]: MIT press, 2022.

DENG, L. The mnist database of handwritten digit images for machine learning research [best of the web]. **IEEE Signal Processing Magazine**, v. 29, n. 6, p. 141–142, 2012.

DJIGAL, H. et al. Ippts: An efficient algorithm for scientific workflow scheduling in heterogeneous computing systems. **IEEE Transactions on Parallel and Distributed Systems**, v. 32, n. 5, p. 1057–1071, 2021.

ELY, A. F. et al. Neat: A scalable soc framework for adaptive neural network acceleration in edge computing. In: **2025 IEEE 16th Latin America Symposium on Circuits and Systems (LASCAS)**. [S.l.: s.n.], 2025. v. 1, p. 1–5.

ELY, A. F. et al. Hara: Hardware-aware resource allocation for fpga-based neural network acceleration. In: **2025 38th SBC/SBMicro/IEEE SBCCI**. [S.l.: s.n.], 2025. p. 1–5.

FRANCO, G.; PAPPALARDO, A.; FRASER, N. J. **Xilinx/brevitas**. Zenodo, 2025. Disponível em: <https://doi.org/10.5281/zenodo.3333552>.

GAREY, M. R.; JOHNSON, D. S. **Computers and intractability: A guide to the theory of NP-completeness**. [S.l.]: WH Freeman, 1979.

GHODRATI, S. et al. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In: **2020 53rd Annual IEEE/ACM MICRO**. [S.l.]: IEEE, 2020. p. 681–697.

GOLDBERG, D. E. **Genetic algorithms in search, optimization, and machine learning**. Reading, Mass.: Addison-Wesley Publishing Company, 1989. xiii, 412 pages : illustrations ; 25 cm p.

GOODFELLOW, I. et al. **Deep learning**. [S.l.]: MIT press Cambridge, 2016. v. 1.

GUO, K. et al. [dl] a survey of fpga-based neural network inference accelerators. **ACM Trans. Reconfigurable Technol. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 12, n. 1, mar. 2019. ISSN 1936-7406. Disponível em: <https://doi.org/10.1145/3289185>.

HOLLAND, J. H. **Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence**. [S.l.]: MIT press, 1992.

JIANG, C. et al. Energy aware edge computing: A survey. **Computer Communications**, Elsevier, v. 151, p. 556–580, 2020.

JORDAN, M. G. et al. Energy-aware fully-adaptive resource provisioning in collaborative cpu-fpga cloud environments. **Journal of Parallel and Distributed Computing**, v. 176, p. 55–69, 2023.

JOUINI, O. et al. A survey of machine learning in edge computing: Techniques, frameworks, applications, issues, and research directions. **Technologies**, MDPI, v. 12, n. 6, p. 81, 2024.

KAO, S.-C. et al. Digamma: Domain-aware genetic algorithm for hw-mapping co-optimization for dnn accelerators. In: **Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe (DATE)**. [S.l.: s.n.], 2022. p. 232–237.

KARABULUT, E. et al. Themis: Time, heterogeneity, and energy minded scheduling for fair multi-tenant use in fpgas. **IEEE Transactions on Computers**, IEEE, 2025.

KERSZ, I. et al. Exploring adaptive design mapping and device activation in multi-fpga edge environments. In: **2025 38th SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.: s.n.], 2025. p. 1–5.

KERSZ, I. et al. Hedgy: Heterogeneous design management for multi-tenant multi-fpga edge systems. In: **2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.: s.n.], 2025. v. 1, p. 1–6.

KERSZ, I. et al. Exploiting design flexibility in multi-tenant multi-fpga edge systems. In: **2025 IEEE 16th Latin America Symposium on Circuits and Systems (LASCAS)**. [S.l.: s.n.], 2025. v. 1, p. 1–5.

KERSZ, I. et al. Adare: Adaptive resource provisioning in multi-fpga edge environments. In: **2024 37th SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.: s.n.], 2024. p. 1–5.

KIM, S. et al. Aurora: Virtualized accelerator orchestration for multi-tenant workloads. In: **Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture**. New York, NY, USA: Association for Computing Machinery, 2023. (MICRO '23), p. 62–76. ISBN 9798400703294. Disponível em: <https://doi.org/10.1145/3613424.3614280>.

KLEINROCK, L. **Queueing systems. Vol. 1, Theory**. New York: Wiley, 1975. xviii, 417 pages : illustrations ; 24 cm p.

KOROL, G. et al. Adaflow: A framework for adaptive dataflow cnn acceleration on fpgas. In: **2022 DATE**. Antwerp, Belgium: IEEE, 2022. p. 244–249.

KOROL, G. et al. Pruning and early-exit co-optimization for cnn acceleration on fpgas. In: **2023 DATE**. [S.l.: s.n.], 2023. p. 1–6. ISSN 1558-1101.

KRIZHEVSKY, A.; HINTON, G. et al. Learning multiple layers of features from tiny images. Toronto, ON, Canada, 2009.

KUON, I.; ROSE, J. Measuring the gap between asic and fpgas. **IEEE Transactions on Computer-Aided Design**, v. 26, p. 203–215, 2007.

KUON, I.; TESSIER, R.; ROSE, J. FPGA architecture: Survey and challenges. **Foundations and Trends in Electronic Design Automation**, v. 2, n. 2, p. 135–253, 04 2008. ISSN 1551-3939. Disponível em: <https://doi.org/10.1561/1000000005>.

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, v. 86, n. 11, p. 2278–2324, 1998.

LI, H. et al. **Pruning Filters for Efficient ConvNets**. 2017. Disponível em: <https://arxiv.org/abs/1608.08710>.

LI, Z. et al. Towards intelligent edge computing: A resource-and reliability-aware hybrid scheduling method on multi-fpga systems. **Electronics**, MDPI AG, v. 14, n. 1, p. 82, 2025.

LIN, W. et al. Soybean image dataset for classification. **Data in Brief**, v. 48, p. 109300, 2023. ISSN 2352-3409. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2352340923004195>.

MANDAL, R.; GOUDA, M. K. Vineyard vigilance: Harnessing deep learning for grapevine disease detection. **J. Emerg. Investig**, 2024.

MAO, Y.; CHEN, X.; LI, X. Max–min task scheduling algorithm for load balance in cloud computing. In: **Proceedings of International Conference on Computer Science and Information Technology**. New Delhi: Springer India, 2014. p. 457–465.

MBONGUE, J. M. et al. Architecture support for fpga multi-tenancy in the cloud. In: **ASAP**. [S.l.: s.n.], 2020. p. 125–132.

MINHAS, U. I. et al. Efficient, dynamic multi-task execution on fpga-based computing systems. **TPDS**, IEEE, v. 33, n. 3, p. 710–722, 2021.

NGUYEN, T. D.; KUMAR, A. Maximizing the serviceability of partially reconfigurable fpga systems in multi-tenant environment. In: **FPGA**. [S.l.: s.n.], 2020. p. 29–39.

RAZAVI, S. A. et al. On exploiting patterns for robust fpga-based multi-accelerator edge computing systems. In: **2022 DATE**. Antwerp, Belgium: IEEE, 2022. p. 116–119.

RUIZ-GUIROLA, D. E.; LÓPEZ, O. L.; MONTEJO-SANCHEZ, S. Modeling iot traffic patterns: Insights from a statistical analysis of an mtc dataset. **Expert Systems with Applications**, Elsevier, v. 272, p. 126726, 2025.

SHI, W. et al. Edge computing: Vision and challenges. **IEEE internet of things journal**, IEEE, v. 3, n. 5, p. 637–646, 2016.

SUBEDI, P. et al. Ai multi-tenancy on edge: Concurrent deep learning model executions and dynamic model placements on edge devices. In: IEEE. **2021 IEEE 14th International Conference on Cloud Computing (CLOUD)**. [S.l.], 2021. p. 31–42.

SUBEDI, P. et al. Ai multi-tenancy on edge: Concurrent deep learning model executions and dynamic model placements on edge devices. In: **CLOUD**. [S.l.: s.n.], 2021. p. 31–42.

TAHERI, M. et al. Deepaxe: A framework for exploration of approximation and reliability trade-offs in dnn accelerators. In: **2023 24th ISQED**. [S.l.: s.n.], 2023. p. 1–8. ISSN 1948-3295.

TANG, Y. et al. Chef: A framework for deploying heterogeneous models on clusters with heterogeneous fpgas. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 43, n. 11, p. 3937–3948, 2024.

TASCI, M. et al. Fpga-qnn: Quantized neural network hardware acceleration on fpgas. **Applied Sciences**, v. 15, n. 2, 2025. ISSN 2076-3417. Disponível em: <https://www.mdpi.com/2076-3417/15/2/688>.

TRIGKA, M.; DRITSAS, E. Edge and cloud computing in smart cities. **Future Internet**, MDPI, v. 17, n. 3, p. 118, 2025.

UMUROGLU, Y. et al. FINN: A framework for fast, scalable binarized neural network inference. In: **FPGA**. New York, NY, USA: ACM, 2017. (FPGA '17), p. 65–74. ISBN 9781450343541. Doi: 10.1145/3020078.3021744.

VIVADO design suite. 2023. Disponível em: <https://www.xilinx.com/products/design-tools/vivado.html>.

XU, C. et al. The case for fpga-based edge computing. **IEEE Transactions on Mobile Computing**, IEEE, v. 21, n. 7, p. 2610–2619, 2020.

YU, F. et al. Clementi: Efficient load balancing and communication overlap for multi-fpga graph processing. **Proc. ACM Manag. Data**, Association for Computing Machinery, New York, NY, USA, v. 3, n. 3, jun. 2025. Disponível em: <https://doi.org/10.1145/3725275>.

ZENG, S. et al. Serving multi-dnn workloads on fpgas: A coordinated architecture, scheduling, and mapping perspective. **IEEE Transactions on Computers**, v. 72, n. 5, p. 1314–1328, 2023.

ZHANG, X. et al. Appq-cnn: An adaptive cnns inference accelerator for synergistically exploiting pruning and quantization based on fpga. **IEEE Transactions on Sustainable Computing**, v. 9, n. 6, p. 874–888, Nov 2024. ISSN 2377-3782.

ZHANG, X. et al. Dnnexplorer: A framework for modeling and exploring a novel paradigm of fpga-based dnn accelerator. In: **2020 IEEE/ACM IICCAD**. [S.l.: s.n.], 2020. p. 1–9. ISSN 1558-2434.

ZOBAED, S. et al. Edge-multiai: Multi-tenancy of latency-sensitive deep learning applications on edge. In: IEEE. **2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)**. [S.l.], 2022. p. 11–20.

ZOBAED, S. et al. Edge-multiai: Multi-tenancy of latency-sensitive deep learning applications on edge. In: IEEE. **2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)**. [S.l.], 2022. p. 11–20.