From Models to Systems: GO HW for Unified AI and Hardware Execution

Whitepaper 1.4
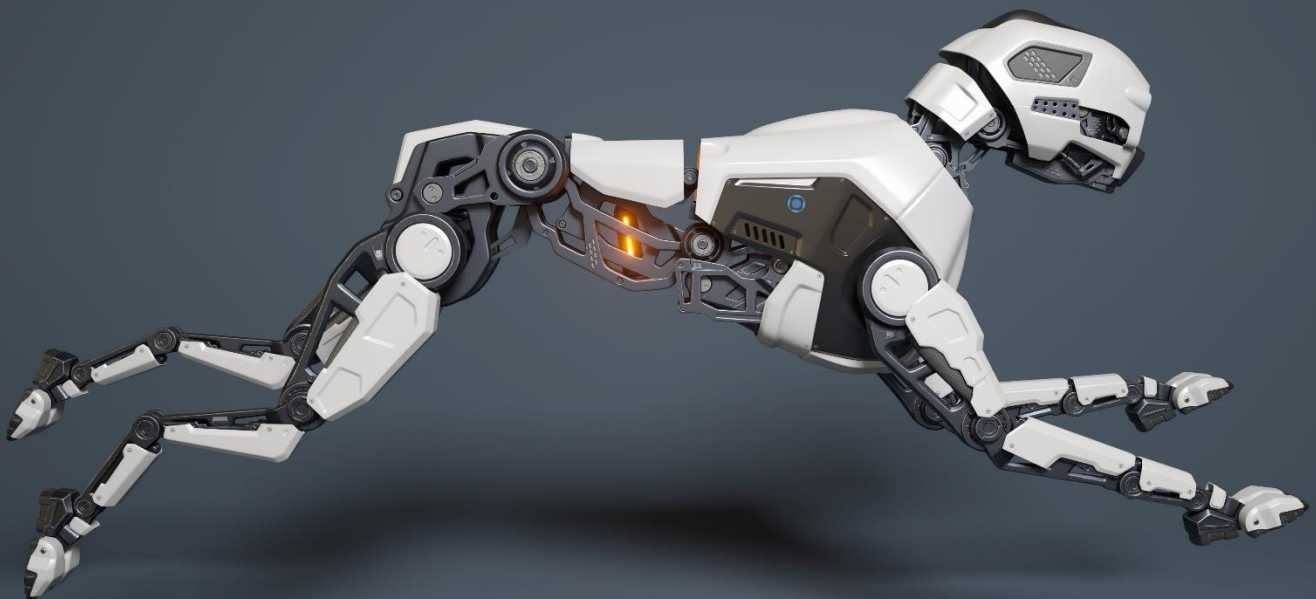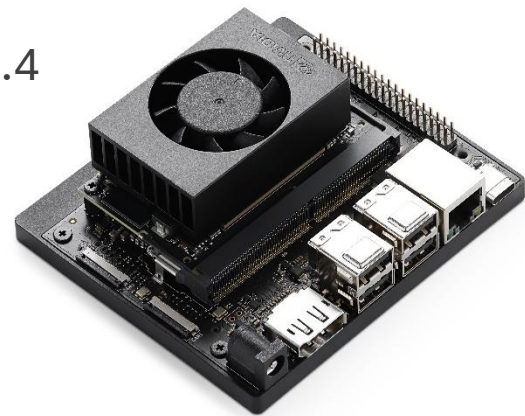
GRAIPHIC

Table of Contents

Versioning

This document is subject to version control to ensure full traceability of changes. Each update is recorded with its author, date, and a short description of the modifications.

Version	Date	Author	Organization	Change Description
1.0	2025/08/31	Youssef Menjour	Graiphic	First publication of the GO HW Whitepaper
1.1	2025/09/04	Youssef Menjour	Graiphic	No SONNX
1.2	2025/09/11	Youssef Menjour	Graiphic	FAQ update
1.3	2025/09/18	Youssef Menjour	Graiphic	IR First / Last
1.4	2025/12/13	Youssef Menjour	Graiphic	Transfer of certain general sections to the SOTA GO general whitepaper
1.4	2025/12/13	Youssef Menjour	Graiphic	Added program-level preface clarifying the role of GO HW within the Graiphic four-whitepaper architecture and linking to the related repositories.

Preface — How This Whitepaper Fits into the Graiphic Program

This whitepaper is part of a coordinated series of four technical documents published by Graiphic to present the full scope, structure, and ambition of its unified AI and graph orchestration program.

Rather than isolated publications, these documents are designed to be read as a coherent whole. Each whitepaper focuses on a specific layer of the same technological stack, while sharing a common foundation built on ONNX, ONNX Runtime, and graphical graph orchestration inside the SOTA ecosystem.

The four whitepapers are:

SOTA — The LabVIEW IDE for Graph Computing: This foundational whitepaper introduces **SOTA**, the first building block of Graiphic's technology stack and a fully functional platform already available to industry, research laboratories, and academic institutions. It is the unified graphical environment developed by Graiphic to author, edit, train, optimize, and deploy ONNX graphs visually inside LabVIEW, and it defines the common execution model, tooling, and orchestration principles used across all other initiatives.

GO HW — From Models to Systems: This whitepaper focuses on extending ONNX-based graphs beyond AI inference into full system and hardware orchestration. It introduces hardware primitives (GPIO, DMA, ADC/DAC, timers, energy monitoring) as first-class graph nodes, enabling deterministic execution of AI-driven systems directly on CPUs, GPUs, FPGAs, NPUs, and embedded SoCs.

GO GenAI — From Fragmentation to Orchestration: This whitepaper addresses the fragmentation of today's Generative AI ecosystem. It transforms ONNX into a dynamic orchestration fabric capable of coordinating models, tokenizers, logic, data streams, and heterogeneous hardware runtimes inside a single executable graph, fully integrated within the SOTA environment and without reliance on Python glue code.

GO IML — From Theory to Superiority: This whitepaper introduces **Informed Machine Learning** as a native graph-based capability. It enables data-driven learning to be augmented with physics, logic, constraints, and expert knowledge directly inside ONNX training graphs, delivering faster convergence, stronger generalization, explainability, and deployability across real-world hardware.

Together, these four documents describe a single, unified vision: **one graph as the source of truth, one runtime for execution, and one graphical cockpit to design, orchestrate, and deploy intelligent systems from models to hardware.**

The complete Graiphic GO Whitepaper Series is available here:
https://github.com/Graiphic/GO-Whitepapers

Executive Summary

ONNX has established itself as the de facto standard for portable AI inference, allowing models to run efficiently across CPUs, GPUs, FPGAs and NPUs. Graiphic's work builds directly on this foundation and extends ONNX into a much broader role: not only a format for inference, but a complete framework for orchestrating AI, logic and hardware in a unified and transparent way.
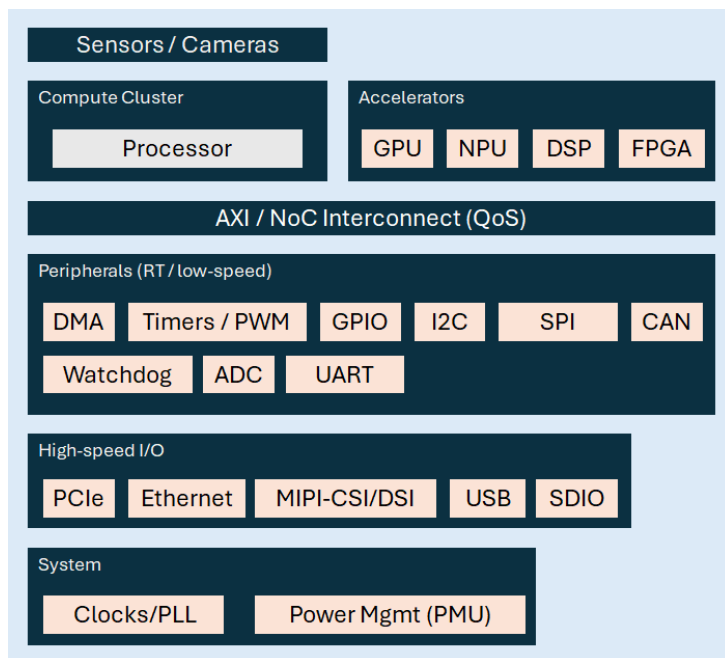
This evolution has unfolded in three major steps. First, we enabled training workflows inside ONNX, combined with LabVIEW orchestration, which are already used in Graiphic's Deep Learning Toolkit. Second, we introduced ONNX GO, an orchestration layer that supports control structures such as conditionals, loops and runtime branching, and which is already deployed in the LabVIEW Accelerator Toolkit. The third step, which this document focuses on, is ONNX GO HW: a new layer that integrates hardware primitives such as DMA transfers, GPIO, ADC/DAC and timers directly into ONNX graphs.

The goal of ONNX GO HW is to make hardware orchestration as seamless and standardized as AI inference itself. The analogy with NI's DAQmx is intentional: just as DAQmx unified hardware configuration and access through a single interface, ONNX GO HW provides an open and portable representation of hardware tasks that can be defined and scheduled inside ONNX graphs. Unlike DAQmx, this approach is not tied to proprietary APIs or devices but remains compatible across multiple runtimes and platforms.

LabVIEW plays a central role as the natural cockpit for this technology. Engineers can visually design, deploy and monitor systems that combine artificial intelligence with real-world hardware control, all within a single workflow. This creates a powerful bridge between abstract AI models and physical systems, with immediate benefits in test and measurement, robotics, industrial automation, aerospace and defense.

ONNX GO HW introduces a new paradigm in execution. By embedding hardware orchestration into standardized graphs, it transforms ONNX from a static description of models into a dynamic and auditable framework capable of managing the entire lifecycle of intelligent systems.

SoC basics without the jargon

System on Chip (SOC) high level design

A System on Chip is a small city on a slice of silicon. You get a CPU for general work, a GPU or NPU for heavy math, memory blocks, and the streets that connect them called buses. Around that city sit the ports that touch the world. General purpose pins switch lights or read buttons. Converters turn voltages into numbers and back. Timers keep time. Interrupts wake the city when something important happens. Put it together and you have a computer that can sense, think, and act on its own.

Think in three layers. At the edge are signals you can touch. GPIO flips digital inputs and outputs. ADC reads analog values like pressure or vibration. DAC writes analog values like a reference voltage. PWM creates precise pulses for motors and LEDs. Timers and counters measure durations and frequencies. Interrupts say stop what you are doing and look here. In the middle is data movement. Direct Memory Access moves blocks of data without bothering the CPU. Small shared buffers act like mailboxes between parts of the chip. At the core sits compute. The CPU runs control logic. The GPU or NPU crunches arrays for vision or language. Caches and formats decide how fast the math flows.

Why does this matter for graphs. Because an ONNX graph can run where the signals originate. A camera feeds a stream into memory. DMA places frames without copies. The runtime reads tensors in place. The decision lands while the belt still moves. Latency drops because you do not ship data across a network. Energy drops because you do not spin big servers for tiny decisions. Portability holds because the same graph can target different SoCs through different providers while keeping the same logic.

Nvidia Jetson Orin SOC

Two pictures make it concrete. In a bottling line a tiny board watches caps and fills. A sensor fires, a frame arrives, a model checks the meniscus, and a reject arm nudges a faulty bottle. The action happens in tens of milliseconds. In a smart street cabinet a board reads weather and traffic sensors, adjusts timing for a crossing, and reports summaries every minute. No one babysits the box. The graph is the script and the chip runs the play.

> *Keep a simple mental kit for SoCs.*
> - *Signals: GPIO, ADC, DAC, PWM, timers, interrupts*
> - *Movement: DMA, shared buffers, ring queues*
> - *Compute: CPU for logic, GPU or NPU for arrays*
> - *Wins: low latency, low energy, same logic on many boards*

State of the Art in Graph Computing and Hardware Orchestration (2021–2025)

Introduction

Graph-based computing has become a cornerstone of modern AI systems. Neural networks are naturally expressed as computational graphs where nodes represent operations and edges represent data flows (tensors). Beyond model inference, many AI workflows – from sensor acquisition to decision-making and actuation – can be modeled as dataflow graphs or directed acyclic graphs (DAGs). Representing workflows in this form enables global optimization, reproducibility, parallelism, and a unified view of the system.

Between 2021 and 2025, major advances have been made in graph compilers, distributed DAG schedulers, hardware-specific runtimes, and pipeline orchestrators. Yet, none of the existing approaches fully unifies **AI computation, orchestration logic, and hardware I/O** under a single portable artifact. This section surveys key academic and industrial efforts and highlights the gap that motivates the development of **ONNX GO HW**.

Graph Compilers and Runtimes

Modern compilers and runtimes transform computation graphs into optimized executables tailored to each hardware target.

- **ONNX Runtime (ORT)** – A cross-platform engine for executing ONNX graphs with kernel fusion, memory planning, and multiple Execution Providers (CUDA, TensorRT, oneDNN, DirectML, OpenVINO, etc.). Widely used in production for portability and performance.
- **Apache TVM** – An open-source compiler stack applying graph-level and tensor-level optimizations, including auto-scheduling (Ansor). Supports CPUs, GPUs, microcontrollers, and custom ASICs.
- **Google XLA / MLIR** – A compiler infrastructure generating optimized HLO IR for CPUs, GPUs, and TPUs, excelling at static graph optimizations.
- **NVIDIA TensorRT** – A high-performance runtime for NVIDIA GPUs, focused on inference, with aggressive optimizations (layer fusion, quantization).

- **Meta Glow** – A graph-lowering compiler producing optimized code for heterogeneous devices, though with declining adoption compared to ORT/TVM.

Strength: excellent inference performance.

Limitation: focus on neural nets only; pre/post-processing, control flow, and hardware orchestration remain external.

Pipeline Orchestration Frameworks

Some frameworks address end-to-end workflows by connecting models with other processing nodes.

- **NVIDIA Triton Inference Server** – Supports ensembles of models connected as DAGs, with batching and scheduling. Optimized for serving at scale, not embedded control.
- **NVIDIA Holoscan (GXF)** – Graph Execution Framework for real-time sensor/AI pipelines on Jetson/Orin. Provides zero-copy buffers and deterministic scheduling, but mainly tied to NVIDIA hardware.
- **NVIDIA DeepStream** – A graph-driven multimedia pipeline framework based on GStreamer, targeting video analytics.
- **ROS 2** – Widely used in robotics, representing systems as graphs of nodes communicating via DDS. Strong for modularity, but determinism and real-time guarantees remain challenging.
- **LabVIEW** – The precursor of visual graph-based programming, with native support for control and I/O. Historically limited by dependence on proprietary runtimes and lack of AI-native integration.

Strength: integration of multiple components (sensing, AI, control).

Limitation: models are often treated as black boxes; no unified graph artifact combining AI and I/O.

Hardware-Specific SDKs and DSLs

Vendors have created specialized graph-oriented SDKs to maximize performance on their chips.

- **AMD Vitis AI** – Compiles models into FPGA DPUs, enabling efficient inference with quantization.
- **Xilinx Adaptive Dataflow (ADF)** – DSL for programming Versal AI Engines as graphs of kernels and streams.
- **Qualcomm QNN SDK** – Constructs hardware-specific graphs for Snapdragon SoCs, mapping to DSPs, NPUs, and GPUs.
- **NVIDIA CUDA Graphs** – API to reduce GPU kernel launch overhead by chaining kernels as graphs.
- **Intel oneAPI / OpenVINO** – Graph IRs optimized for Intel CPUs, GPUs, and VPUs.

Strength: hardware efficiency, near-ASIC performance.
Limitation: vendor lock-in; portability and orchestration across vendors not supported.

Limitations of the Current State of the Art

Despite the breadth of solutions, several gaps remain:

1. **Fragmentation** – Inference engines, orchestrators, and hardware SDKs remain siloed, requiring glue code.
2. **No unified artifact** – AI models, control loops, and I/O logic are spread across different runtimes.
3. **Vendor lock-in** – Each hardware vendor exposes its own graph DSL, reducing portability.
4. **Lack of determinism** – Few frameworks address real-time guarantees, safety profiles, or certifiability for aerospace/automotive/defense.

Transition to GO HW

These gaps open the path for **ONNX GO HW**:

- One **graph artifact** (ONNX) for computation, orchestration, and I/O.
- One **runtime engine** (ONNX Runtime) that schedules both math and hardware nodes.
- One **cockpit** (LabVIEW-style IDE) to author, configure, deploy, and monitor.

This unified approach is portable across vendors, auditable for safety-critical domains, and efficient for embedded deployment.

Comparative Snapshot

Framework / SDK	AI Models	Control Flow	Hardware I/O	Portability	Real-Time / Safety
ONNX Runtime	Yes	Limited (If/Loop)	No	High	Partial (fallbacks)
Apache TVM	Yes	No	No	High	No
TensorRT	Yes (GPU)	No	No	NVIDIA-only	No
ROS 2	Yes (as external node)	Yes	Yes (via drivers)	High	Limited determinism
Holoscan / DeepStream	Yes	Partial	Yes (streams)	NVIDIA-only	Some deterministic scheduling
Vitis AI / QNN / ADF	Yes	No	Partial	Vendor-only	Limited
GO HW (proposed)	Yes	Yes (If/Loop/Scan)	Yes (GPIO, DMA, ADC/DAC, PWM)	High (via Execution Providers)	Yes

Closing Sentence

The fragmentation of today's tools highlights the need for a unified solution. **This gap motivates GO HW, a concrete path from static description to dynamic technology.**

GO HW, a concrete path from static description to dynamic technology

GO HW stands for Graph Orchestration for Hardware. It turns a single ONNX graph into a living control loop that runs on real chips. The artifact stays ONNX. The engine stays ONNX Runtime. The cockpit stays LabVIEW. What changes is that hardware primitives become first-class nodes and timing becomes part of the plan.

Take the thought experiment and give it a name. GO HW is our way to turn ONNX from a static file into a living plan. ONNX stays the language that describes the graph. ONNX Runtime stays the engine that compiles and executes the plan. LabVIEW stays the cockpit where people think in graphs. GO is the glue that makes schedules, policies, and lifecycle first class citizens inside the same artifact.

The change is simple to feel. Instead of juggling scripts and private formats, teams keep one ONNX bundle that carries model structure, training logic when needed, evaluation flows, and housekeeping such as metrics and checkpoints. The runtime sees the whole plan, fuses what it can, sizes memory once, and delivers a session that behaves the same every time you call it. Reviews become graph diffs. Tests become graph runs. Rollbacks become file swaps.

We do not invent a new file or a new engine. We make better use of what exists. Control flow nodes like If, Loop, and Scan are not extras for edge cases. They are the handles that let you encode learning loops, curriculum choices, early stopping, and reporting without leaving the ONNX world. The result is a clean pipeline that is portable, auditable, and friendly to both engineers and researchers.

Ergonomics remains a first-class concern. Layer mode gives practitioners the speed of Keras style building blocks. Node mode gives researchers fine control at the operator level. Both write to the same ONNX graph. Both compile to the same session. Both benefit from the same runtime optimizations. Your team chooses the view. The artifact stays one.

> *Three promises of GO HW*
> - *One artifact for the lifecycle*
> - *One engine for performance and portability*
> - *One cockpit that makes graphs natural to author and reason about*

GO HW on SoCs, author, configure, deploy, monitor

Think of GO HW as a four-step groove. You author a graph, you configure a target, you deploy a compiled plan, you monitor the run. Same artifact, same engine, different boards.

Author. Build the model as an ONNX graph in the LabVIEW cockpit. Engineers use layer blocks when they want speed. Researchers switch to node level when they want

precision. Pre and post processing live in the same graph when it makes sense. Shapes, dtypes, and opset are checked early so the file is clean before you touch hardware.

Configure. Pick a board and load its device profile. The profile describes memory, supported providers, and practical limits such as how many concurrent streams make sense. The tool suggests a partitioning plan across providers. You confirm what runs on CPU, what runs on GPU or NPU, and what the Hardware EP will bind when hardware primitives are present. One click produces a plan you can review.

Deploy. ONNX Runtime compiles the graph into a session for that SoC. Kernels that fit together are fused. Memory arenas are sized to avoid hot path allocations. Formats are aligned so tensors do not ping pong between layouts. You ship a compact bundle that contains the graph, the compiled artifacts, and a small manifest. The device starts the session and keeps it resident.

Monitor. A tiny agent speaks gRPC for a side channel. Operators can read metrics, watch a few tensor taps, and adjust whitelisted parameters without touching the hot path. The session keeps its timing and memory stable. The agent handles logs, health checks, and safe restarts. You can stage a new graph, switch over, and roll back with short commands.

Here is a simple way to keep the loop healthy.

> *Happy path checklist*
> - *Validate shapes and opset at author time*
> - *Compile once per target and reuse the session*
> - *Keep preprocessing and metrics in the graph when practical*
> - *Track versions with a clean semantic tag and a short changelog*
> - *Warm up the session after boot and record a baseline*

A small example makes it real. A sorter runs on a Jetson today and moves to a Zynq board next quarter. You do not rewrite the logic. You open the same ONNX file, select a different device profile, compile for the new target, and deploy. ONNX Runtime picks the right providers. The agent reports the same metrics. Operations do not learn a new tool. They keep their eyes on the same cockpit.
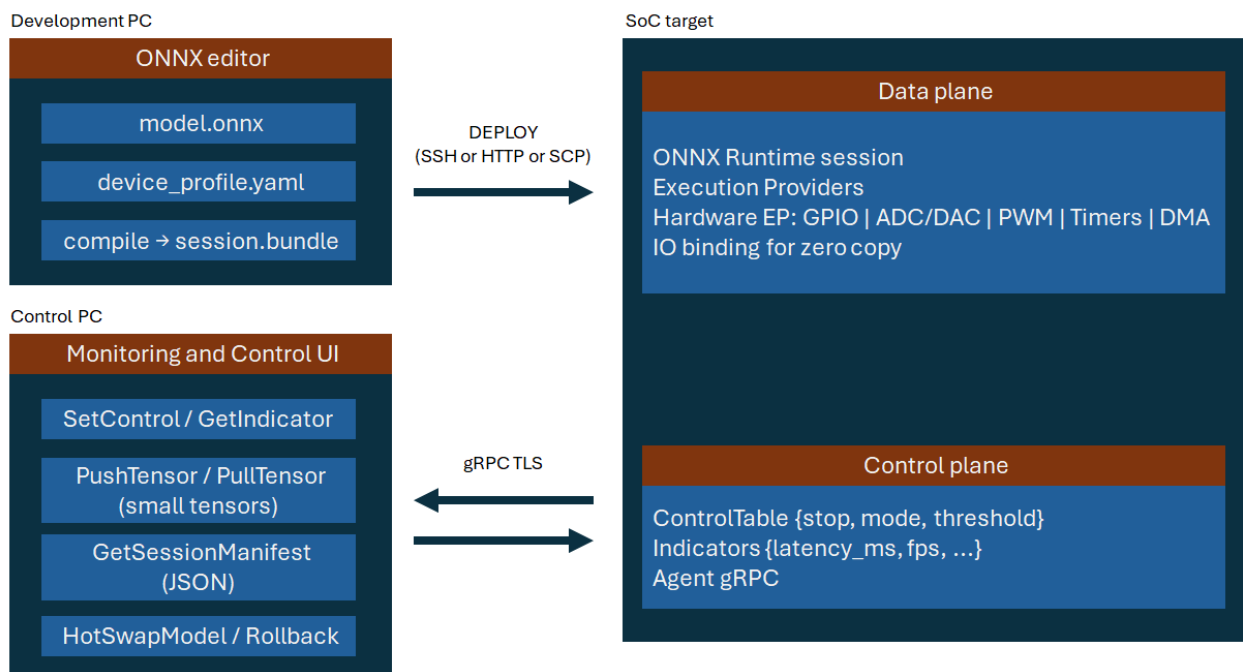
First functional architecture for GO HW

The model graph authored on the development PC is encapsulated as a subgraph within a larger control graph, wrapped in a Loop (while) node with a configurable cadence and an exit condition set in the editor.

On the Development PC you work in the ONNX editor inside LabVIEW. You author the graph as model.onnx, pick a device_profile.yaml for the target, then compile to a session.bundle. The bundle contains the optimized graph, the chosen Execution Providers, and the I O bindings the runtime will use. You deploy this bundle to the SoC over SSH or HTTP or SCP.

On the SoC target the data plane runs an ONNX Runtime session. Execution Providers accelerate subgraphs on the available engines. A Hardware EP exposes GPIO, ADC and DAC, PWM, timers and DMA as graph accessible services. I/O binding connects device or pinned buffers to inputs and outputs so tensors move without extra copies. Beside the data plane sits the control plane. It keeps a small ControlTable for parameters like stop, mode and threshold, an Indicators set for states like latency and fps, and a gRPC agent that exposes these knobs. The control plane never blocks the hot path. DMA is kept for large streams such as camera or audio, while small controls use local registers.

On the Control PC the Monitoring and Control UI speaks four simple verbs over gRPC with TLS. SetControl and GetIndicator write and read small parameters and states. PushTensor and PullTensor send or fetch small tensors through named bindings, useful for calibration or checks. GetSessionManifest returns a JSON snapshot of the active session with providers, partitions and I O bindings so you can inspect what runs where.

HotSwapModel replaces the active bundle after validation and warmup, and Rollback restores the previous one if a health check fails.

This layout keeps one artifact, one engine and one cockpit. Today the LabVIEW loop still drives the session tick by tick, which makes debugging simple. Next we bring more schedule into the graph with If, Loop and Scan while keeping the same planes, the same API and the same zero copy path.

Proposed hardware nodes (first wave)

- **sys.ControlGet(name)** → read a small scalar/vector from the ControlTable (e.g., threshold, mode, stop).
- **sys.IndicatorSet(name, value)** → publish metrics/states (latency_ms, fps, temperatures).
- **sys.Clock(period | source)** → provide ticks or timestamps to cadence a Loop.
- **sys.TriggerIn(source)** → edge or level trigger from external signal or timer.
- **sys.Delay(ms | cycles)** → bounded delay inside a control subgraph.
- **sys.MetricTap(tensor, rate)** → sample a tensor safely for monitoring without perturbing the hot path.
- **sys.Watchdog(timeout_ms, safe_action)** → enforce a safe state if timing budgets are missed.
- **sys.SafeState(action)** → explicit fallback action (e.g., de-energize outputs).
- **hw.GPIOIn(pin)** → boolean/u8 input; debouncing as attribute.
- **hw.GPIOOut(pin, value)** → digital output with optional pulse attributes.
- **hw.ADCIn(chan, shape)** → acquire analog samples into a tensor; sampling rate as attribute.
- **hw.DACOut(chan, value)** → write analog value; optional slew/limits.
- **hw.PWMOut(chan, duty, freq)** → generate PWM; jitter and range as attributes.
- **hw.RTFifoDequeue(name) / hw.RTFifoEnqueue(name, tensor)** → bounded real-time queues for small deterministic streams.
- **hw.DMARead(channel, bytes|shape)** → zero-copy intake from device to tensor (camera, ADC DMA).
- **hw.DMAWrite(channel, tensor)** → zero-copy out to device.

Binding rules. Controls/Indicators live in local RAM (small scalars). RT-FIFO for small deterministic streams. DMA is reserved for large flows. All bindings are **named handles** (no raw pointers) and appear in the session **manifest**.

Once a graph touches real pins, safety and evidence stop being optional. We introduce observability and device profiles so teams can trust what runs and prove it.

Energy-Aware Graphs and Forensic Monitoring

A missing cornerstone of today's AI deployment is energy. Performance metrics such as latency and accuracy dominate the discussion, yet energy – the joules consumed per

inference or training step – remains invisible. GO HW enables users to explicitly elevate energy as a **scientific, reproducible metric** inside graph orchestration. Models can thus be evaluated not only for their predictions, but also for their execution cost, with energy becoming part of the same first-order evidence as accuracy or latency.

GO HW extends the monitoring plane with energy measurement. Each Execution Provider can expose an optional Energy Provider API to start and stop sampling during graph execution. Readings from hardware counters (e.g. NVML/PCAT for NVIDIA GPUs, RAPL for Intel CPUs, INA3221 for Jetson SoCs, PMBus for FPGA boards) are collected and normalized into joules per run. While these sources provide useful indicative values, they do not always reach **forensic-level precision**. To address this gap, Graiphic envisions building dedicated **test benches per hardware platform**, equipped with calibrated external instrumentation, and publishing the results openly in the same transparent manner as its Execution Providers Tester project. This approach ensures that hardware profiles are backed by auditable, high-precision evidence of energy consumption.

These measurements become **energy semantics**: annotations attached to nodes, subgraphs, or sessions, preserved in the Session Manifest. They enable reproducibility (same model, same joules), comparability (different boards, same metric), and accountability (evidence for audits and certification).

Beyond monitoring, GO HW introduces a **new family of loss functions where energy is a first-class component**. Users may define multi-objective optimization goals ($L = \alpha *$ $error + \beta * joules$), or construct custom losses directly from measured values by wiring hardware counter nodes into the training graph. This allows energy to be treated as a **standard optimization signal**, not merely as an external log. Graph-level optimizations such as kernel fusion, quantization, pruning, and early exit branches further reduce consumption without altering hardware.

By combining indicative monitoring, forensic test benches, semantic annotation, and energy-aware loss design, GO HW makes energy visible, actionable, and reproducible in AI systems deployed on real hardware.

LabVIEW-native forensic measurement.

A defining advantage of Graiphic's GO HW project is its native LabVIEW environment. LabVIEW has long been the gold standard in test and measurement, and this DNA translates directly into energy-aware AI orchestration. Beyond relying on low-level counters (temperature sensors, CPU utilization, or memory load), GO HW can leverage LabVIEW instrumentation to build rigorous test benches for each target SoC. These benches combine calibrated DAQ hardware with reproducible orchestration scripts, enabling precise measurements of joules consumed per model, per architecture, and per workload.

In practice, this means that GO HW can go beyond inference from indirect indicators and provide forensic-grade energy profiles. These results can be benchmarked systematically across Raspberry Pi, Jetson, Zynq, i.MX and industrial IPCs, then published openly on Graiphic's GitHub as reference datasets. By doing so, Graiphic not only monitors energy but sets a reproducible standard for the entire community, where performance is always reported together with energy consumption. This approach ensures transparency, comparability, and long-term credibility, aligning with DARPA's ambition to make energy a first-class scientific metric in machine learning.

Open benchmarking and transparent culture.

A key part of Graiphic's DNA is a culture of transparency and open collaboration. We systematically publish our benchmarks and tools on GitHub, not only to demonstrate capability but also to provide the community with actionable insights. A representative example is the [Execution Providers Tester](#), an open-source initiative that systematically maps ONNX Runtime operator coverage across all Execution Providers. This project, maintained as part of SOTA, has become a reference point for developers and vendors to understand backend support, prioritize missing operators, and track reproducibility across environments.

We apply the same philosophy to energy. With GO HW, our goal is to build and publish forensic-grade energy benchmarks per SoC, validated with LabVIEW-native test benches and external instrumentation. These results will be openly shared on Graiphic's GitHub, in the same transparent manner as our operator coverage tester. By doing so, Graiphic not only demonstrates mastery of the entire ONNX stack, from operators to orchestration to hardware execution, but also provides the ecosystem with clear, reproducible metrics and actionable objectives. This open benchmarking culture ensures trust, comparability, and alignment with DARPA's emphasis on scientific rigor.

Algorithmic Enhancements: Dynamic Loss Functions and Informed Learning through Full Graph Orchestration

Dynamic, energy-aware loss design.

Unlike conventional frameworks where loss functions are hard coded into the training loop, GO HW, built on Graiphic's SOTA orchestration layer, treats the loss as a first-class graph component. This enables the injection of dynamic loss subgraphs at runtime, blending traditional accuracy-driven objectives with hardware-derived energy metrics exposed via new HW nodes such as GPIO, DMA, timers, ADC/DAC, and power counters. Losses can therefore explicitly minimize both prediction error and joules consumed. For example: $L = \alpha \times error + \beta \times energy$

This capability is unique. SOTA is currently the only framework that can dynamically orchestrate and reconfigure such hybrid objectives directly inside the ONNX graph, making energy optimization a native part of the training loop rather than an afterthought.

Graph-compilation efficiency as orchestration property.

GO HW leverages ONNX Runtime Execution Providers, which means that every graph passes through optimization pipelines where redundant operations are removed and compatible nodes are fused into efficient kernels. What differentiates Graiphic's approach is that these compiler-level passes are orchestrated, inspected, and controlled at the graph level. Energy gains are no longer incidental side effects of compilation; they are visible, reproducible orchestration choices. Only a framework with full graph mastery such as SOTA can expose and standardize this capability.

Integration of alternative learning paradigms.

Through its orchestration-first architecture, GO HW seamlessly integrates self-supervised and informed learning approaches as native graph constructs. In self-supervised learning, contrastive vision methods such as SimCLR have exceeded supervised ImageNet performance with **100 times fewer labels**, while in NLP masked token prediction achieves state-of-the-art results with minimal annotation. In informed learning, domain constraints and physical laws are encoded directly into the optimization graph. Zhang et al. (2021) demonstrated that an elastic-energy-constrained network matched supervised accuracy without ground-truth data. In computational fluid dynamics, informed networks reproduced full simulations approximately **60 times faster** than FEM or FVM solvers. In structural engineering, PINNs incorporating conservation laws provided more precise stress predictions while reducing computational cost. These paradigms converge faster, require fewer epochs, and consume significantly less energy. With GO HW's orchestration layer, they become deployable as graph-native strategies rather than external workarounds.

Impact: Green AI by design.

By uniting dynamic energy-aware losses, orchestrated compiler optimizations, and alternative ML paradigms, GO HW transforms energy from a passive metric into an active design variable. This positions Graiphic's SOTA as the only fully orchestrated graph framework able to embed energy directly into learning, ensuring AI systems that are not only accurate but also efficient, reproducible, and sustainable.

Energy-aware contributions of SOTA and GO HW

Category	SOTA contributions	GO HW contributions
Prevention (reduce energy upfront)	▪ Graph orchestration allows dynamic injection of energy-aware loss functions. ▪ Operator fusion, pruning, quantization embedded at graph level. ▪ Support for alternative paradigms (self-supervised, informed ML) reducing training epochs and data labeling effort. ▪ Training orchestration inside ONNX graphs (loss, optimizer, control flow) avoids scattered scripts, ensuring leaner execution.	▪ Hardware primitives (GPIO, DMA, timers, ADC/DAC) exposed as nodes, enabling energy-aware design directly tied to SoC resources. ▪ Cross-hardware portability (CPU, GPU, FPGA, SoC) allows selecting the most energy-efficient target. ▪ Automatic graph compilation with kernel fusion and memory planning across Execution Providers to reduce wasted cycles.
Monitoring (measure and expose energy)	▪ Native LabVIEW environment provides intrinsic test & measurement DNA. ▪ Session manifests and metrics are integrated into orchestration layers for reproducible runs. ▪ Early monitoring hooks for latency, memory, and resource usage.	▪ Forensic-grade energy monitoring via Execution Providers extended with Energy Provider APIs (NVML, RAPL, INA3221, PMBus). ▪ Dedicated LabVIEW test benches with calibrated DAQ for per-SoC joule measurement. ▪ Energy semantics preserved as annotations in Session Manifest. ▪ Open GitHub culture: publishing per-model/per-architecture benchmarks for transparency (e.g. Execution Providers Tester precedent).
Curation (optimize after deployment)	▪ Ability to re-train or fine-tune models by dynamically adjusting losses including energy terms. ▪ Graph-level rewrites (kernel fusion, pruning, early exits) as corrective strategies. ▪ Switch between inference/training/academic sessions for flexible post-hoc tuning.	▪ Migration of trained models across hardware (GPU → FPGA/SoC) to achieve better energy/performance trade-offs. ▪ Hot-swap and rollback mechanisms in deployed sessions without breaking timing or safety constraints. ▪ Multi-objective optimization functions (error + joules) guiding iterative refinement of deployed systems.

Closing, one graph, many roles

The path is simple to state. Keep one artifact. Let it describe models, training schedules, and workflows. Run it with one engine across many devices. Give people a cockpit that feels natural. This is how ONNX, ONNX Runtime, and the LabVIEW experience come together in GO HW.

What changes for teams is focus. You spend less time stitching scripts and more time shaping graphs. You review optimized graphs like code. You move the same file from experiment to evaluation to deployment. You target a workstation, a Jetson, a Zynq, or a PC, and the logic stays the same. You gain speed because the runtime plans the work. You gain trust because the plan is visible.

What changes for the ecosystem is reach. A native ONNX editor removes the dependency on third party export paths. Control flow nodes become everyday tools instead of hidden features. The SONNX safety profile gives sensitive sectors a clear contract for meaning and evidence. ONNX grows from an excellent file format to a complete graph computing framework that spans learning, serving, and control.

Implementation and Deployment of ONNX GO HW on SoCs (Raspberry Pi 5 as First Case Study)

Raspberry PI 5
Broadcom BCM2712
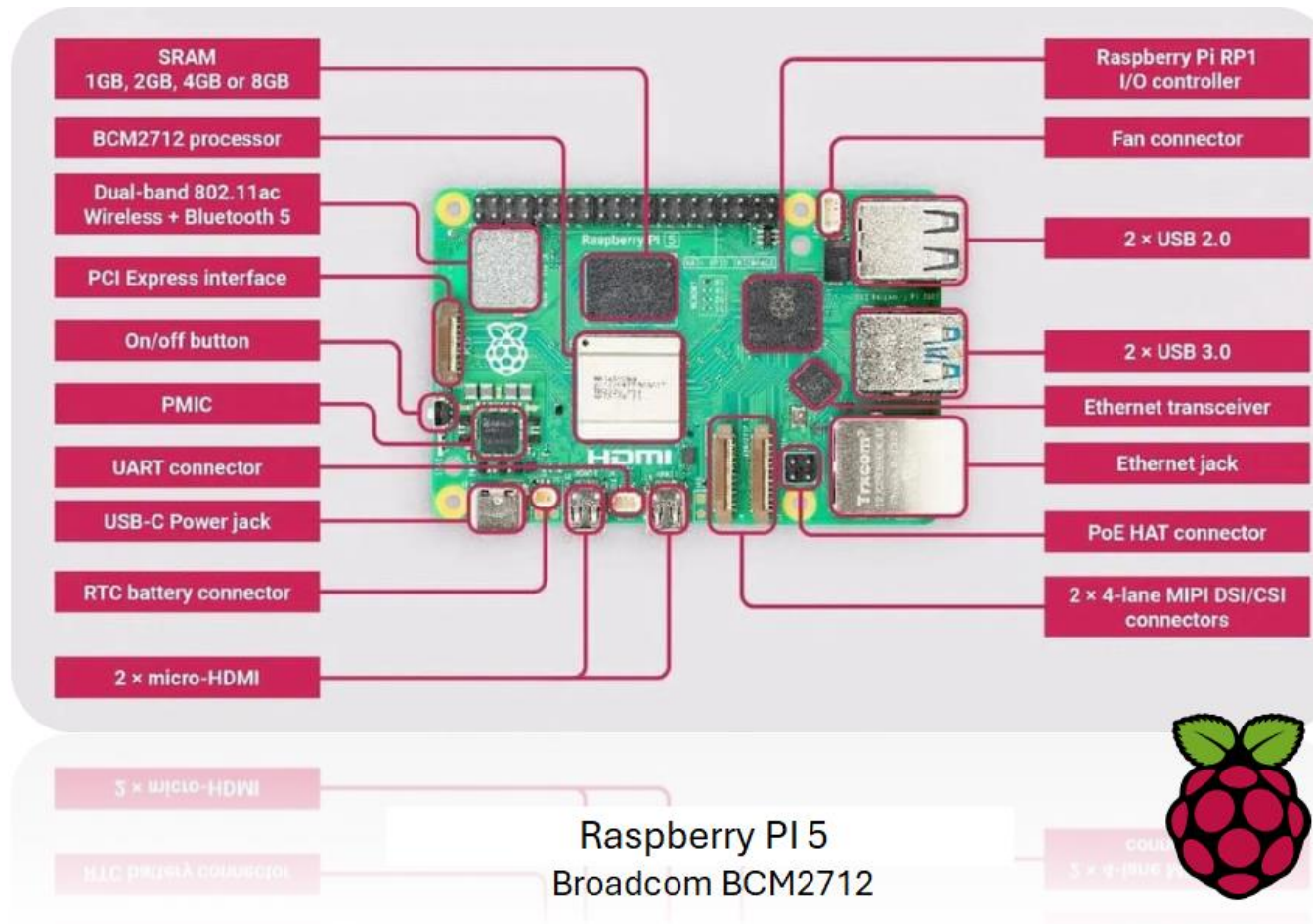
The implementation of ONNX GO HW follows a structured workflow, moving from operator specification to deployment and monitoring on real SoCs. We start with the **Raspberry Pi 5** (Figure 0) as the initial demonstrator. Based on the **Broadcom BCM2712**, this board provides a rich set of interfaces (USB, Ethernet, GPIO, PCIe, etc.), making it an ideal target to define and validate the initial architecture.

On the development side, three core artifacts are essential:
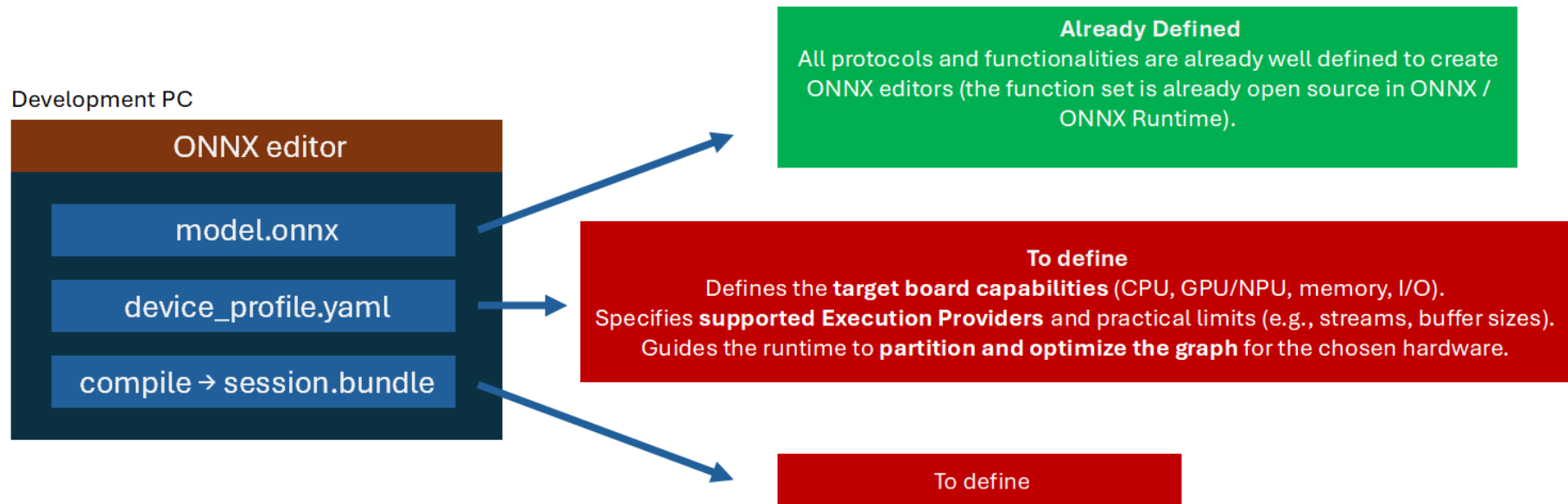
- **model.onnx**: the graph definition, representing the universal contract.
- **device_profile.yaml**: the SoC "identity card," describing available resources (CPU, GPU/NPU, memory, I/O) and constraints.
- **session.bundle**: the optimized execution package ready for deployment on the target. These artifacts build on existing ONNX / ONNX Runtime mechanisms, ensuring interoperability and standardization.

A first proof-of-concept is already visible in the LabVIEW **Graiphic IDE,** which demonstrates that custom development environments can be constructed directly from the open-source ONNX toolchain. This highlights the possibility for third-party IDEs to leverage the same functional core.
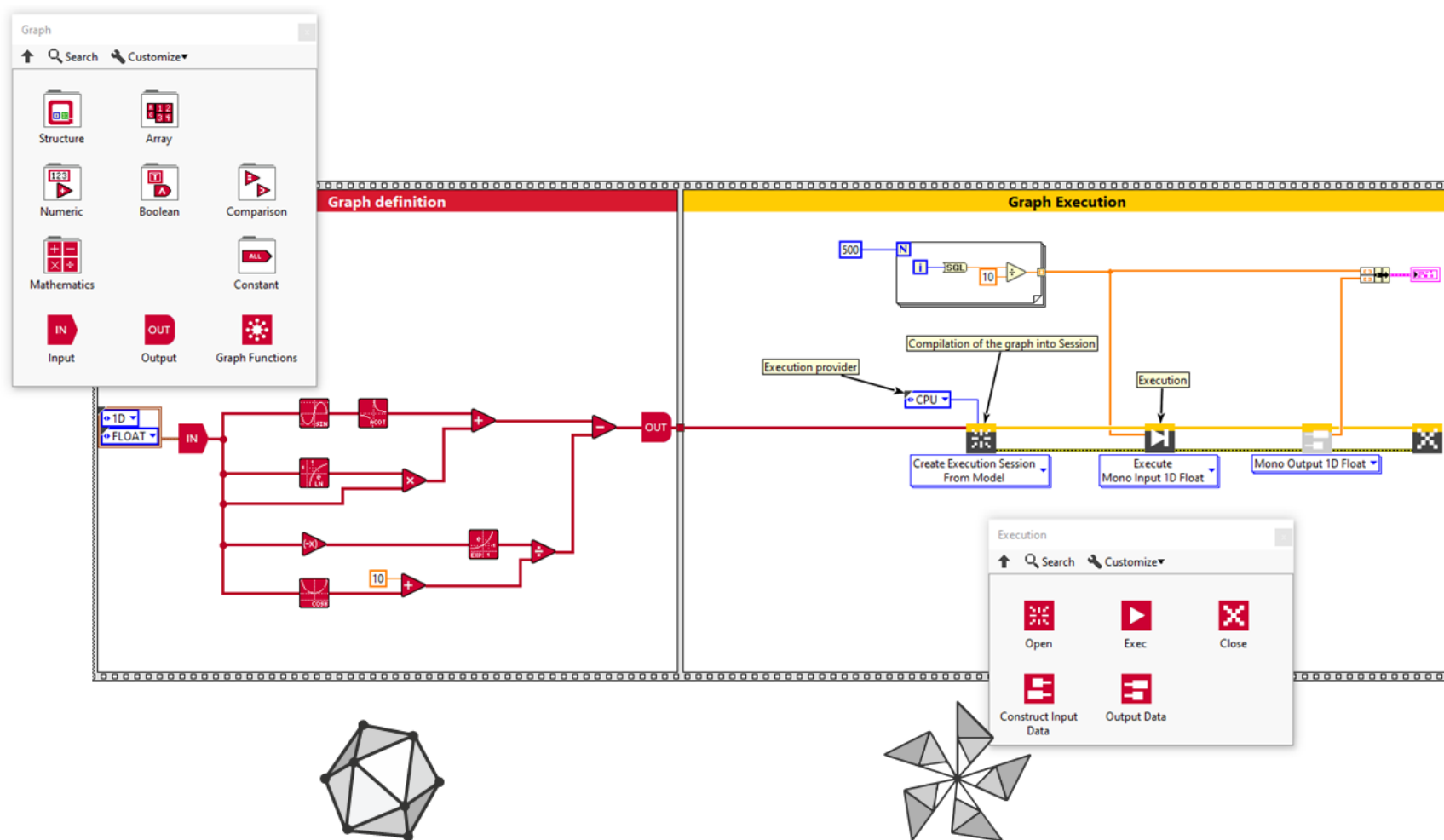
The development workflow can be divided into two categories:

1. **Already defined by ONNX** (graph compilation, operator schemas, execution management).
2. **To be specified** (hardware node encapsulation, standardized SoC profiles).

At deployment time, two execution scenarios are possible:
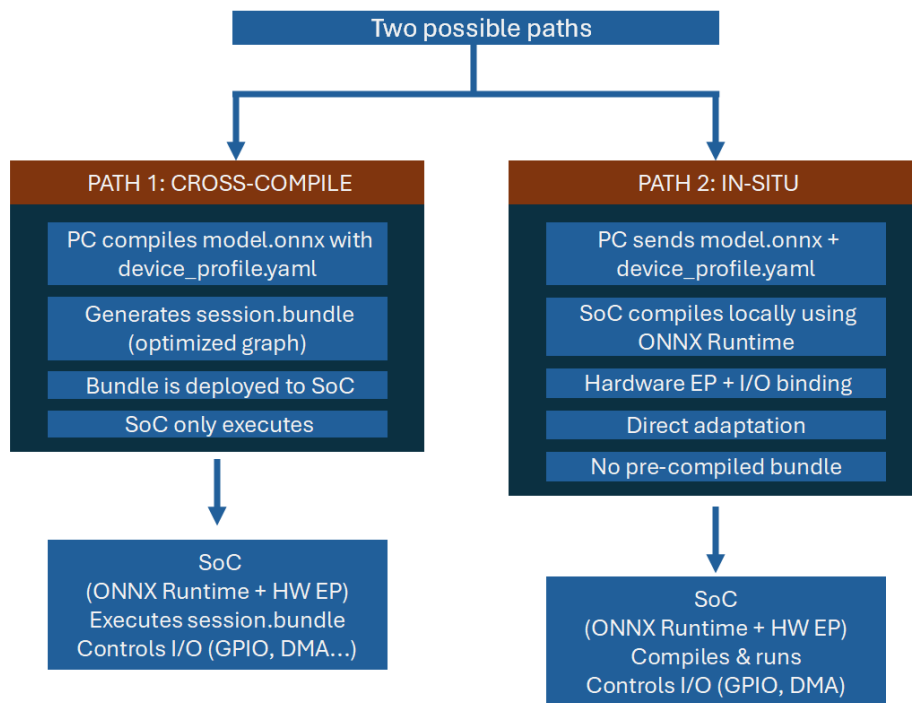
- **Cross-Compilation**: the graph is compiled on the development PC using the target profile, producing an optimized **session.bundle** that is transferred to the SoC. Advantage: simulation can be performed before deployment.

- **In-situ Compilation**: the PC sends the model and profile, and ONNX Runtime compiles directly on the SoC, adapting natively to available hardware resources.

Once deployed, the architecture runs inside the **SoC execution environment** (Figure 5):

- **Data Plane**: ONNX Runtime session extended with Hardware Execution Providers (GPIO, ADC/DAC, PWM, DMA, etc.), using optimized memory bindings for zero-copy execution.
- **Control Plane**: runtime configuration via a control table (start/stop, thresholds, modes), performance indicators (latency, FPS, metrics), and a gRPC-based agent ensuring monitoring, remote access, and execution safety.

23

Finally, the **supervision layer** establishes secure communication via gRPC TLS between the **Control PC** and the SoC. It exposes a standardized set of functions: configuration and indicator retrieval, small tensor transfers, access to the session manifest (JSON), and dynamic model lifecycle management (HotSwap/Rollback).

SoC target

To define
Create encapsulate
architecture for SOC

DEPLOY
(SSH or HTTP or SCP)

Data plane

ONNX Runtime session
Execution Providers
Hardware EP: GPIO | ADC/DAC | PWM | Timers | DMA
IO binding for zero copy

To define
First Architecture for a
RasberyPI5 (to start)

To define
Create gRPC agent within
SOC software architecture

gRPC TLS

Control plane

ControlTable {stop, mode, threshold}
Indicators {latency_ms, fps, ...}
Agent gRPC

Conclusion

Demonstrating this workflow on Raspberry Pi 5 will provide the first **end-to-end validation** of ONNX GO HW, proving that ONNX can orchestrate SoC-level resources through an optional hardware namespace. From this initial feasibility study, the experience gained will drive:

- **standardization** of SoC profiles and hardware nodes,
- **Progressive extension** to additional platforms (Jetson, Zynq, FPGA, i.MX, etc.),
- **reproducibility** through portable device_profile.yaml descriptions and unified functions.

This approach ensures that each new SoC can be integrated into the ONNX GO HW ecosystem through the addition of a profile in a shared library, guaranteeing **interoperability, portability, and long-term adoption**.

Proposed Path Forward for ONNX steering committee

To ensure clarity and avoid fragmentation, we propose the creation of an **experimental optional domain** under ONNX, tentatively named **onnx.hardware**.

- **Status**: The operators defined in this domain would be *optional*, exactly like other ONNX operators that are not universally implemented across Execution Providers. No vendor would be required to support them.
- **Scope**: Initial focus on SoC primitives such as GPIO, DMA transfers, ADC/DAC, timers, and synchronization nodes.
- **Deliverables**:
    1. Draft operator definitions with schemas and documentation.
    2. Mapping tables to existing vendor APIs (e.g., CUDA, DAQmx, XRT, OpenVINO, oneAPI).
    3. An open-source prototype (starting with Raspberry Pi 5) demonstrating feasibility.
- **Governance**: This domain would be managed under a new **ONNX Hardware Working Group**, working in coordination with existing WGs (e.g., Multi-Device, Generative AI).
- **Goal**: Provide a recognized namespace where hardware orchestration operators can be incubated in a structured way, ensuring legitimacy, community visibility, and long-term alignment with the ONNX standard.

This approach guarantees that ONNX remains lightweight at its core while providing a credible framework for vendors and industrial adopters who wish to expose hardware-level capabilities within ONNX graphs.

For teams adopting GO HW

- *Treat the ONNX file as the single source of truth*
  Version it and generate everything else from it.
- *Keep preprocessing and metrics in the graph when practical*
  Improves reproducibility and portability.
- *Compile once per target and reuse sessions for stable timing*
  Preallocate, warm up, and keep sessions resident.
- *Use Registers for safe live tuning and monitoring*
  Clear names, bounded values, agent-side validation and ACLs. Prefer handles over raw addresses; enable debug mode only on bench rigs.
- *Enable Manifest and Log with the right level*
  Verbose in dev, minimal in prod, alert on thresholds.
- The story began with a simple idea. A graph can be more than a snapshot of a model. It can be the plan that a runtime turns into reliable behavior on real machines. It can be the language that engineers and researchers share. It can be the bridge between design and the world.

Target platform matrix for GO HW v1.0

Below is a pragmatic, coherent set of targets where GO HW will focus first. Each line shows typical ONNX Runtime providers and the hardware-node bindings expected.

Embedded SoCs and SBCs
- Raspberry Pi 4 and 5 Providers: CPU with XNNPACK or ACL when available. Notes: Linux, GPIO and PWM via kernel drivers, SPI/I2C/ADC through HATs, DMA for high-rate streams where supported.
- NVIDIA Jetson family (Orin, Xavier, Nano) Providers: CUDA, TensorRT. Notes: CSI cameras through DMA, GPIO and PWM via libgpiod, high-bandwidth video and DNN offload.
- AMD Xilinx Zynq UltraScale+ and Kria Providers: Vitis AI where available, CPU fallback. Notes: FPGA fabric for deterministic I O, DMA engines, RT-FIFO patterns, PWM and timers in PL or PS.
- NXP i.MX 8M PlusProviders: CPU with XNNPACK, optional NPU via vendor EP when available. Notes: Industrial-friendly I O, camera pipelines, decent power envelope.
- TI Sitara AM62/AM64 Providers: CPU with XNNPACK. Notes: PRU-based I/O timing, EtherCAT on selected SKUs, good for control loops.
- Rockchip RK3588 class boards Providers: CPU with XNNPACK, vendor NPU EP where supported. Notes: Strong CPU, plentiful I O, popular in edge boxes.

Industrial PCs and controllers
- Intel x86-64 IPCs Providers: OpenVINO for CPU and iGPU, CPU EP as baseline. Notes: Rich PCIe and field-bus cards, predictable thermal budget, long-term support.
- Windows IPCs with discrete or integrated GPUs Providers: DirectML for compatible GPUs, CPU EP as baseline. Notes: Useful where Windows tooling is mandatory.

Industrial automation vendors for device profiles and field-bus drivers
- Beckhoff Focus: EtherCAT device profile, GPIO/PWM mapping, DMA paths on IPCs.
- Siemens Focus: Industrial PCs and edge devices, Profinet profiles, OPC UA bridges.
- Schneider Electric Focus: Industrial PCs, Modbus and Ethernet/IP profiles, gateway patterns.
- Gantner Focus: high-precision DAQ profiles, synchronized sampling and streaming.

This list keeps GO HW general and vendor-neutral. It concentrates on Linux-capable SoCs and IPCs where ONNX Runtime already runs well and where hardware nodes can bind cleanly to GPIO, timers, FIFOs, and DMA. Classic closed PLC runtimes are out of scope. Integration happens through industrial PCs or Linux controllers that sit next to PLCs and talk over field buses.

Frequently Asked Questions (FAQ)

Q1. What happens if a hardware node is used but the target device does not support it?
A1. The behavior will be defined by the Graiphic Hardware Working Group. Possible policies include:
Error: execution stops with a clear diagnostic.
Fallback: a safe software emulation or CPU path is used.
The choice will be discussed and standardized openly, so the community agrees on the expected semantics.

Q2. What if a hardware platform cannot execute the requested operation due to limitations?
A2. Again, the Working Group will define the policy. Options include a strict error or an explicit fallback to a compatible provider. The key is that the behavior is not left to private implementations, it is decided transparently by the community to ensure predictability and reproducibility.

Q3. Does GO HW force hardware vendors to implement new operators?
A3. No. Just like ONNX Runtime today, where Execution Providers do not support every ONNX operator (and this has never been a blocker), GO HW operators are optional. Vendors can freely decide which hardware nodes to support.
To ensure transparency, Graiphic has already benchmarked this phenomenon through the **Execution Provider Coverage Tester** (another Graiphic-led open source initiative), which documents operator support across EPs. The same principle naturally applies to ONNX HW: optional nodes, no obligation, and clear visibility of what each vendor supports.

System / Versions

- **Test Date:** 2025-08-29 10:49:37
- **CPU:** Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz
- **GPU:** NVIDIA GeForce RTX 2070
- **ONNX:** 1.18.0 | **ONNX Runtime:** 1.23.0+cu125
- **ONNX Opset:** 20 | **ONNX IR:** 10

ONNX Core Operators

Execution Provider	SUCCESS	FALLBACK	SUPPORTED	FAIL	NOT TESTED	SKIPPED	TRAINING
CPU	149 (96%)	0 (0%)	149 (96%)	4 (3%)	0 (0%)	0 (0%)	41 (27%)
Intel - OpenVINO™	91 (59%)	56 (36%)	147 (95%)	6 (4%)	0 (0%)	0 (0%)	0 (0%)
Intel - oneDNN	36 (23%)	113 (73%)	149 (96%)	4 (3%)	0 (0%)	0 (0%)	0 (0%)
NVIDIA - CUDA	100 (65%)	49 (32%)	149 (96%)	4 (3%)	0 (0%)	0 (0%)	39 (25%)
NVIDIA - TensorRT	87 (56%)	60 (39%)	147 (95%)	6 (4%)	0 (0%)	0 (0%)	0 (0%)
Windows - DirectML	113 (73%)	35 (23%)	148 (95%)	5 (3%)	0 (0%)	0 (0%)	0 (0%)

Microsoft Custom Operators

Execution Provider	SUCCESS	FALLBACK	SUPPORTED	FAIL	NOT TESTED	SKIPPED	TRAINING
CPU	59 (55%)	0 (0%)	59 (55%)	41 (38%)	7 (7%)	0 (0%)	7 (7%)
Intel - OpenVINO™	15 (14%)	41 (38%)	56 (52%)	44 (41%)	7 (7%)	0 (0%)	0 (0%)
Intel - oneDNN	6 (6%)	52 (49%)	58 (54%)	42 (39%)	7 (7%)	0 (0%)	0 (0%)
NVIDIA - CUDA	52 (49%)	34 (32%)	86 (80%)	14 (13%)	7 (7%)	0 (0%)	6 (6%)
NVIDIA - TensorRT	6 (6%)	78 (73%)	84 (79%)	16 (15%)	7 (7%)	0 (0%)	0 (0%)
Windows - DirectML	27 (25%)	33 (31%)	60 (56%)	40 (37%)	7 (7%)	0 (0%)	0 (0%)

Q4. Why not just leave this as custom operators managed by individual companies?
A4. Without a common framework, every vendor would create incompatible extensions, leading to fragmentation and lock-in. A Working Group ensures that policies, schemas, and naming are agreed upon openly and democratically, avoiding duplication and incompatibility across the ecosystem.

Q5. How can we address heterogeneous scenarios like DMA, given that each hardware vendor follows its own protocol?
A6. The reasoning is the same as in ONNX Runtime today: Execution Providers are not identical and often have different behaviors. ONNX provides a common abstraction layer while allowing vendor-specific implementations. In practice, there will always be a shared denominator (common patterns across hardware) and vendor-specific details. The ONNX HW Working Group will be the place to define:
- how to parameterize nodes in a generic way,
- how to expose vendor-specific extensions,
- and how to manage exceptions consistently.

This ensures both interoperability and flexibility, without forcing uniformity across all vendors.

Call for Funding: Why Industry Should Invest in GO HW

Graiphic has built the first end-to-end ecosystem where AI, logic, and hardware orchestration live inside a single ONNX graph.

We are now opening a Call for Funding to accelerate the roadmap of GO HW.

Why Invest?

- Strategic Advantage: Gain early access to the first universal cockpit that unifies AI + hardware orchestration across CPUs, GPUs, FPGAs, NPUs, and SoCs.
- Portability & Standards: Ensure your hardware, SDKs, and platforms are natively supported in a framework that is becoming the de facto open standard.
- Energy & Efficiency: Join the revolution of Green AI by design. GO HW introduces forensic-grade energy metrics and optimization directly inside ONNX graphs.
- Safety & Trust: Participate in shaping the SONNX safety profile, unlocking adoption in aerospace, defense, automotive, healthcare, and critical infrastructure.
- Market Reach: From Raspberry Pi to NVIDIA Jetson, from industrial PLCs to cloud servers, GO HW runs everywhere, and your technology can be part of it.

What We Offer

- Co-development opportunities with our engineering team.
- Early integration of your platforms and SDKs into GO HW.
- Joint visibility in international standardization efforts (ONNX, DARPA, Horizon Europe, ADRA).
- Shared benchmarking and open-source dissemination to establish your technology as a leader in AI orchestration.

How to Engage

Graiphic is actively seeking:

- Equity investors ready to support our growth.
- Industrial sponsors willing to co-fund R&D and test benches.
- Strategic partners (hardware vendors, system integrators, large OEMs) who want their platforms at the heart of the future ONNX Hardware ecosystem.

Join us in shaping the universal cockpit for AI.

Contact: funding@graiphic.io | www.graiphic.io

Annexes

Support Letters

Emerson
11500 North Mopac Expressway
Austin, TX 78759 United States
Phone: (877) 388-1952

July 2025

Subject: Support for Graiphic's ONNX-GO Hardware Application

To Whom It May Concern,

On behalf of Emerson, we would like to express our support for Graiphic's ONNX-GO Hardware initiative. As the creator of LabVIEW, a graphical programming platform widely used in industry, research, and education, Emerson recognizes the value of expanding LabVIEW's capabilities to address the next generation of intelligent and connected systems.

ONNX-GO Hardware will create the first unified execution infrastructure enabling AI inference, training, signal processing, and low-level hardware control to run inside a single dynamic ONNX graph on heterogeneous System-on-Chip (SoC) platforms. The initiative will enable real-time and auditable orchestration of mission-critical pipelines.

We believe that such an initiative can strongly position LabVIEW as a versatile tool for building and orchestrating advanced intelligent systems, and we look forward to following its progress and exploring potential opportunities for collaboration.

Sincerely,

Kevin Schultz
Chief Technology Officer
Test and Measurement Group
Emerson

RAM Aviation, Space & Defense
3172 E. Deseret Dr
St. George, UT 84790, USA

August 28, 2025

Subject: Support for Graiphic's ONNX Hardware Project

To Whom It May Concern,

I am writing on behalf of RAM Aviation, Space & Defense (RAM ASD) to express our strong support for the proposal to incubate the ONNX Hardware Working Group (ONNX HW) under the ONNX Steering Committee, as detailed in Graiphic's GO HW Whitepaper.

As a leading designer and manufacturer of electro-mechanical devices serving the aviation, space, and defense sectors, RAM ASD operates in environments where unified AI and hardware orchestration is essential for mission-critical applications. We frequently encounter challenges stemming from the fragmentation of vendor-specific SDKs and the lack of standardized abstractions for hardware primitives such as GPIO, DMA, timers, and ADC/DAC. This leads to inefficiencies in development cycles, increased energy consumption, reduced inference performance, and difficulties in achieving reproducibility across diverse SoCs like Raspberry Pi, Jetson, and Zynq platforms.

The proposed ONNX HW Working Group directly addresses these pain points by establishing a dedicated hardware namespace, an operator set for SoC primitives, and protocols for graph creation, hardware flashing, and orchestration. By integrating seamlessly with ONNX Runtime through Execution Providers, this initiative will extend ONNX beyond static inference to a comprehensive framework for AI + logic + hardware orchestration. We see tremendous value in its potential to deliver cross-platform reproducibility, published capability matrices, and open-source outputs under the MIT license, all while ensuring long-term stewardship by Graiphic and community contributions.

In our industry, where demands for efficient, real-time systems in robotics, industrial automation, and aerospace/defense are rapidly growing, ONNX HW will help mitigate

the risks of fragmentation and accelerate ONNX adoption. It aligns with our goals of developing portable, high-performance solutions that minimize wasted cycles and enhance system reliability.

RAM ASD fully endorses this proposal and is committed to supporting its success. We look forward to engaging with the Working Group through validation, feedback, or potential contributions, and we are confident that ONNX HW will drive innovation and efficiency across the ecosystem.

Should you require any additional information or clarification, please do not hesitate to contact me.

Sincerely,

Timothy Cannon *Tim Cannon*
Control System Engineer
RAM Aviation, Space & Defense
Phone: 435-772-8681
Email: tim.cannon@ramasd.com

Graph Computing for AI Systems: State-of-the-Art (2021–2025)

Introduction

Graph-based computing has become fundamental in modern AI systems. Neural networks are naturally expressed as computational **graphs** – nodes represent operations or layers and edges represent data flows (tensors) between them. Likewise, complex AI pipelines (from sensor I/O to model inference to actuator control) can be modeled as **directed acyclic graphs (DAGs)** or dataflow programs. Representing AI workflows as graphs enables global optimizations, parallelism, and clarity in system orchestration[1][2]. This review surveys recent advances (2021–2025) in graph computing for AI, covering both academic research and industrial frameworks. We examine computational graph execution, dataflow/DAG systems, graph neural networks, and runtime scheduling on heterogeneous hardware, as well as major graph-centric AI frameworks. We then compare these solutions to the emerging ONNX GO HW approach for unified, real-time AI system orchestration.

Academic Advances in Graph Computing (2021–2025)

Computational Graph Execution: Static vs Dynamic Graphs and Scheduling

Early deep learning frameworks like TensorFlow (<=1.x) employed **static computational graphs**, requiring the full network graph to be defined (and optimized) ahead of execution. Newer frameworks such as PyTorch and TensorFlow 2.x emphasized **dynamic graphs** – networks defined imperatively, allowing flexible structures (e.g. loops, conditionals) and easier debugging[3]. **Static graphs** excel at global optimization: the graph can be compiled for efficient execution (node fusions, memory planning, etc.), often yielding faster runtimes once built. **Dynamic graphs** offer flexibility for dynamic control flow and

variable-length data, at some cost to peak performance. Modern systems are increasingly **hybrid**. For example, PyTorch 2.0 introduced TorchDynamo and other JIT compilers to capture dynamic graphs and compile them, aiming to get the best of both worlds (eager flexibility with optimized execution)[1]. Recent research also explores new scheduling algorithms for computational graphs. Zhao *et al.* (OSDI 2023) present a method to **effectively schedule DNN computation graphs** on specialized accelerators by co-designing with hardware architecture, achieving over **11× speedup vs. TVM** on a custom domain-specific chip[4][5]. Overall, the state-of-the-art emphasizes graph compilers and schedulers that can optimize computation graphs end-to-end, maximizing hardware utilization while accommodating dynamic behaviors.

Dataflow and DAG-Based Systems for Distributed and Embedded Execution

Beyond neural network graphs, many AI workflows are orchestrated as **dataflow pipelines** or DAGs, especially in distributed or edge deployments. Data preprocessing, model inference, and postprocessing can be chained as graph nodes. Academic systems like **Ray** and **DAG-aware schedulers** aim to efficiently distribute such task graphs over clusters, but embedded and real-time settings pose additional constraints (latency, memory). **Flow-based programming** concepts have re-emerged in AI: for instance, streaming systems (Apache Beam/Google Dataflow) represent computations as DAGs of operations that can scale out. On the embedded side, research has examined combining control and AI in **signal processing pipelines** using DAG scheduling with real-time constraints[6][7]. Many robotics and IoT applications use graph-based **pipe-and-filter** architectures: nodes for sensing, perception (AI models), planning, and actuation, connected by data streams. Ensuring **deterministic execution** and low jitter in these DAGs is an ongoing challenge. Academic works on real-time DAG scheduling on heterogeneous CPUs/accelerators (e.g. for autonomous vehicles) have introduced approaches like *graph scheduling with GNN-based heuristics*[8], indicating a crossover of graph computing and learning-based optimization. In summary, representing AI system workflows as dataflow graphs is now common; recent research is improving how we map and schedule these DAGs across distributed or embedded resources for efficiency and reliability.

Graph Neural Networks and Graph-Based Data Processing

In parallel to using graphs for computation scheduling, AI models themselves increasingly operate on graph-structured **data**. **Graph Neural Networks (GNNs)** have become a major research frontier, extending deep learning to arbitrary graph data (social networks, molecules, knowledge graphs, etc.). GNN research 2021–2025 produced new architectures and theoretical insights. Early GNNs like GCN and GraphSAGE used localized message-passing; more recent models incorporate **attention and transformer mechanisms** to capture long-range dependencies on graphs[9]. For example, **Graphormer** (Microsoft, 2021) demonstrated that with the right positional encodings, a Transformer can achieve state-of-the-art on graph benchmarks by effectively treating the graph as a fully-connected attention network[9][10]. There is also growing work on **temporal GNNs** (graphs that evolve over time) – a 2023 survey formalized the state-of-the-art and open challenges in temporal graph learning[11]. Moreover, the **scalability** of GNNs is a key focus: techniques like neighbor sampling, mini-batch training, and distributed GNN frameworks (e.g. DGL, PyTorch Geometric) enable learning on large

graphs with millions of nodes. Researchers are exploring high-performance GNN training on CPU-GPU clusters[12][13] and even custom hardware (e.g. Graphcore IPUs specialized for graph workloads). Another trend is **combining GNNs with causal inference and knowledge graphs**, or using GNNs in multi-agent systems and program analysis[14][15]. In summary, graph-based data processing is now a staple of AI, and state-of-the-art GNN techniques push both **model accuracy** and **efficiency** (with co-design from algorithms down to hardware acceleration[16][17]).

Runtime Systems and Heterogeneous Scheduling for Computation Graphs

Executing computational graphs efficiently on **heterogeneous hardware** (CPUs, GPUs, FPGAs, NPUs, etc.) is an active research area. Academic work has shown that graph compilers and runtimes can drastically improve performance by optimizing placement, memory layout, and kernel fusion. For instance, Furutanpey *et al*. (2025) comprehensively evaluated **neural network graph compilers** across hardware and found that vendor-specific optimizations (TensorRT, OpenVINO, etc.) can *invert* which model runs faster on a given hardware[18][19]. This underscores that compilers are now as important as model architecture for deployment. Modern compilers like **TVM** (Apache TVM) use auto-tuning to generate optimized kernels for each target, achieving **performance portability** across diverse backends[20]. Similarly, Google's XLA (used in JAX and TensorFlow) and Meta's **Glow** compiler perform graph-level optimizations (constant folding, operator fusion) and emit device-specific code. A key development is support for **graph partitioning and offloading** – splitting a graph so parts run on specialized accelerators (e.g. DSP, FPGA) while others run on CPU/GPU. Research on scheduling such partitions shows that considering hardware topology (memory hierarchy, interconnect) when partitioning yields big gains[4][21]. There is also interest in **real-time scheduling** of neural network graphs on accelerators in safety-critical contexts[22]. A recent survey (2023) on real-time scheduling for accelerators notes the need for deterministic execution of computation graphs under timing constraints[23]. In summary, state-of-the-art runtime systems use graph-level knowledge to orchestrate execution across heterogeneous hardware, achieving major throughput improvements while beginning to address predictability and real-time needs.

Industrial Frameworks and Ecosystems

Modern AI software stacks heavily leverage graph representations. Below we review major industrial frameworks in three categories: **model compilers/runtimes**, **graph-based pipeline orchestrators**, and **hardware-specific graph SDKs/DSLs**.

Model Compilers and Runtimes (Graph Optimizers)

- **ONNX Runtime (ORT)** – A high-performance, cross-platform engine for executing ONNX computational graphs. Developed by Microsoft, ORT takes an **interoperable graph** (ONNX model) and optimizes it with graph rewrites and kernel fusions, then dispatches to hardware-specific backends (EPs). It provides a flexible API and integrates many **hardware accelerators** via *Execution Providers*, from CUDA and TensorRT to DirectML and CoreML[24][25]. ORT is widely used in production for its portability – the *same* ONNX graph can run on CPU, GPU, mobile NPUs, etc., with the runtime choosing the fastest path[26].

- **Apache TVM** – An open source deep learning compiler stack that builds end-to-end optimized code for models. TVM takes a model's graph (from frameworks like PyTorch or TensorFlow) and applies optimizations at both the graph level (operator fusion, layout changes) and the **tensor operation level** (auto-tuned kernel code generation)[20]. It aims for *performance portability*: developers write a model once, and TVM can compile it for CPUs, GPUs, ASICs, and even microcontrollers[27]. Techniques like the Ansor **auto-scheduler** explore optimized compute schedules automatically[28]. TVM has become a backbone for many vendor-specific compilers and is used in Amazon's and ARM's toolchains for efficient inference.

- **Google XLA** – The Accelerated Linear Algebra compiler, originally for TensorFlow, now underlies JAX and parts of PyTorch (via TorchXLA). XLA traces and **compiles whole computation graphs** into optimized executables (HLO IR) for each target (CPU, GPU, TPU). It excels at **static graph** optimizations – constant folding, operation fusion, buffer reuse – and can also do ahead-of-time compilation for production. XLA's graph optimizations improve performance and can give more predictable execution (important in Google's large-scale deployments).

- **NVIDIA TensorRT** – A high-throughput deep learning inference runtime that optimizes neural network graphs for NVIDIA GPUs. TensorRT performs aggressive optimizations like combining layers, using reduced precision (FP16/INT8), and auto-tuning kernels. It represents the model as a graph of layers and integrates with NVIDIA's CUDA libraries. In practice, TensorRT often significantly reduces latency and increases throughput for CNNs, transformers, etc. on GPU. However, it is limited to NVIDIA hardware and primarily focuses on inference (not training).

- **Meta Glow** – A graph lowering compiler from Facebook (Meta) that targets various hardware backends. Glow takes in a neural network computation graph and lowers it through two IR levels: an **optimization IR** for high-level graph opts, and a **lower-level IR** closer to hardware ops[29]. It can then generate code for CPUs, GPUs, or custom accelerators. Glow's design emphasizes a modular backend architecture and has been used to deploy models on mobile devices and specialized ASICs at Meta[30]. (Glow is open source, though in recent years ORT and TVM have seen broader adoption.)

- **Others** – *OpenVINO* (Intel) is another graph-oriented runtime, converting models to an IR and optimizing for Intel CPUs, iGPUs, and VPUs[19]. *TensorFlow Lite* uses FlatBuffer graphs and delegation to hardware drivers for mobile inference. PyTorch's *NNAPI* and *CoreML* backends similarly convert the PyTorch graph to run on Android or iOS accelerators. All these frameworks share the goal of maximizing model execution efficiency via graph-level insights, at the cost of additional compilation or conversion steps.

Graph-Based Pipeline Orchestration Systems

- **NVIDIA Triton Inference Server** – A server framework for deploying AI models at scale, with support for **model pipelines** (ensembles). Triton treats an *ensemble* as a DAG of models and processing steps, where the output of one model feeds the next[31]. This allows building end-to-end AI services (e.g. decode image → detect objects → filter results) all within the server. The ensemble DAG execution is handled by Triton's scheduler, avoiding extra data copies and network hops

between models[31][32]. Triton supports multi-framework models (TensorFlow, PyTorch, ONNX, etc.) and handles scheduling, batching, and I/O, making production deployment of graph-based AI pipelines easier.

- **NVIDIA Holoscan & GXF** – **Holoscan** is an SDK for real-time sensor and AI processing on edge devices (e.g. NVIDIA Jetson/Orin). It is built on the **Graph Execution Framework (GXF)**, which executes component networks with strict scheduling. In GXF/Holoscan, an application is described as a *compute graph* of **entities** (nodes) connected by edges[33]. Each entity contains components (codelets) for specific tasks, and the framework provides a scheduler, memory manager (for zero-copy buffers), and message passing primitives[34][35]. Developers can string together sensor input nodes, AI inference nodes, and output nodes, and Holoscan will orchestrate them with low latency. This is used in domains like medical imaging, where a stream of data must pass through an AI pipeline on-device in real time.

- **NVIDIA DeepStream** – A graph-driven pipeline framework specialized for video analytics and IoT, built on GStreamer. DeepStream allows construction of vision processing pipelines (ingest streams, decode, infer with DNNs, track, display) using a **graph specification**. Under the hood, each GStreamer element (e.g. a decoder, an inference plugin) is wrapped as a node (component) in a graph, managed by NVIDIA's graph composer runtime[36][37]. This lets developers assemble complex multimedia AI applications with minimal coding – the configuration (often via YAML or a visual tool) defines how frames flow through a DAG of plugins. DeepStream, coupled with Graph Composer, thus exemplifies a graph-based orchestrator for edge AI, though primarily targeting NVIDIA hardware and streaming use cases.

- **ROS 2 (Robot Operating System 2)** – A popular open-source framework for robotic systems, which follows a **dataflow graph** paradigm at a high level. A ROS 2 system is composed of many modular *nodes* (sensing, planning, control, etc.) that exchange messages via *topics* (or services), forming a computational graph of the robot's software[38]. ROS 2's middleware (DDS) handles message transport between nodes, which may be distributed across multiple processors. The *ROS graph* can be introspected (e.g. via *rqt_graph*) to see how data flows between components. While ROS 2 is not limited to AI, it increasingly integrates AI modules (for example, a node running a deep learning model subscribing to camera images and publishing detections). It provides a standardized way to orchestrate complex, distributed systems, but being a general framework, achieving hard real-time behavior or deterministic scheduling requires additional patterns or the Real-Time ROS extensions.

- **LabVIEW** – An established graphical programming environment (from National Instruments) based on the **dataflow model**. Engineers "program" by connecting functional blocks (nodes) with wires (edges that carry data), naturally creating a graph that represents the system logic. LabVIEW has been traditionally used for instrumentation, control, and measurement systems. Each loop, formula, or I/O operation is a node in a *LabVIEW block diagram*, and the LabVIEW runtime schedules execution according to dataflow: a node runs when all its inputs have data available[39]. This approach made complex systems easier to design and visualize. However, deploying LabVIEW programs onto embedded targets

historically required the LabVIEW runtime or FPGA-specific code generation, limiting portability. LabVIEW is a precursor of modern DAG orchestration tools in its philosophy, and its visual programming style remains highly accessible. Recent efforts (e.g. from Graiphic) even integrate AI model graphs (via ONNX) into LabVIEW, blurring the line between traditional dataflow programming and AI graph execution[40][41].

Hardware-Specific SDKs and Graph DSLs

- **AMD Vitis AI** – A comprehensive development stack for accelerating AI inference on Xilinx/AMD FPGAs and adaptive SoCs. **Vitis AI** includes model optimizers, quantization tools, and compilers that take a trained network (TensorFlow, PyTorch, ONNX, etc.) and compile it to run on a FPGA's deep learning processing unit (DPU) IP. It is essentially a graph compiler + runtime specialized for Xilinx devices. Developers can deploy models on edge boards (like Zynq, Versal) with support for 8-bit quantization and batch processing. According to AMD, *"Xilinx Vitis AI is a development stack for AI inference on Xilinx hardware platforms"*, allowing integration of one or more DPU accelerator kernels into a design[42]. The stack provides APIs to run the compiled model and manage memory, with the goal of near ASIC-like efficiency using reconfigurable logic.

- **Adaptive Dataflow (ADF) – Xilinx Graph DSL for AI Engines**: Xilinx's Versal ACAP platforms include an array of VLIW processors called **AI Engines (AIE)**. These are programmed via the **ADF API** (Adaptive Data Flow), a C++ graph DSL. Using ADF, developers specify a **dataflow graph of kernels** (functions) and streams connecting them, which the toolchain then schedules onto the many-core AIE array[43][44]. Kernels exchange data via ping-pong buffers (windows) or streams over the on-chip network. The ADF model lets multiple kernels execute in parallel, streaming data between each other without going to off-chip memory. For example, a signal processing pipeline of filters and neural network layers can be mapped as a graph, and the ADF runtime ensures each kernel runs on an AI Engine core with synchronized data movement. This graph-level programming is crucial to fully harness the AIE fabric's performance, and it abstracts away a lot of the low-level thread and DMA management for the programmer[43].

- **Qualcomm QNN (AI Engine Direct SDK)** – Qualcomm provides the **Qualcomm Neural Network (QNN) SDK**, also known as AI Engine Direct, for running AI models on Snapdragon SoC accelerators (Hexagon DSPs, NPUs, GPUs). Developers or frameworks (like ONNX Runtime) use QNN to construct a **hardware-specific graph** from an abstract model, which can then be executed on the device's AI cores[25]. For instance, the ONNX Runtime QNN execution provider transforms an ONNX model into a QNN graph and delegates it to Qualcomm's libraries[25]. The QNN SDK handles heterogeneous execution across CPU, Adreno GPU, and the Hexagon-based **HTP (Hexagon Tensor Processor)**. It includes offline quantization and optimization tools as well. QNN is essentially Qualcomm's answer to TensorRT or Vitis: it optimizes neural network graphs to leverage specialized DSP/NPU instructions for fast inference on mobile/embedded platforms.

- **Other Graph DSLs/SDKs** – *NVIDIA CUDA Graphs* (introduced in CUDA 10) allow forming a graph of GPU kernels and memcopy operations to reduce launch

overhead – useful for regular inference/training loops. *Intel oneAPI/dnnl* graph (formerly nGraph) was an IR to represent deep learning computations for Intel accelerators, now part of OpenVINO. *ARM NN* SDK provides a graph-based API to run networks on ARM CPU, Mali GPU, or Ethos NPU. Many smaller vendors (Cambricon, Imagination, etc.) also expose graph-level compilers to integrate their neural accelerators. The common theme is exposing a graph abstraction to developers so that the runtime can map computations efficiently to the hardware, rather than writing device-specific code for each layer.

Toward Unified Graph Orchestration: ONNX GO HW vs. Existing Solutions

Despite the rich ecosystem above, current solutions often address **pieces** of the AI system puzzle. Each framework tends to focus either on neural network computation or on pipeline orchestration or on low-level hardware acceleration, but not all at once. This fragmentation means engineers stitch together multiple tools – for example, using TensorRT for model inference inside a ROS2 or LabVIEW application for control logic, and writing custom glue code for I/O and scheduling. Below, we compare what existing frameworks do well and where they fall short, especially in light of goals like full-system graph unification and real-time performance.

Strengths of Existing Frameworks: Modern compilers and runtimes excel at **optimizing neural network graphs for performance**. They provide tremendous speed-ups by fusing operations and leveraging hardware-specific libraries (for instance, graph compilers can "enhance throughput by orders of magnitude" without changing model accuracy[1]). Meanwhile, pipeline tools like ROS 2 and DeepStream are great at **modularity and integration** – they break complex systems into nodes and allow mixing and matching components (sensors, AI models, etc.) via standardized interfaces[38]. Industrial orchestrators handle concurrency and data transport (e.g. Triton's ensemble scheduler avoids overhead by keeping data on-device between model stages[31][45]). Hardware-specific SDKs provide **maximal efficiency** on their targets – e.g. Vitis AI can get FPGA inference running with low batch latency, and QNN squeezes optimal performance from Snapdragon NPUs. In summary, each class of tool is highly optimized for its domain: neural network execution, multi-component pipelines, or low-level hardware utilization.

Limitations and Gaps: No current framework fully **unifies AI models, general logic, and I/O control in one graph** with portability across systems. Graph compilers (ORT, TensorRT, etc.) treat the model in isolation – any surrounding logic (preprocessing, decisions based on model output, device commands) must be implemented in separate code. Pipeline orchestrators like ROS or LabVIEW handle I/O and control flow but typically treat the AI model as a black box or external function call, rather than integrating it into a single executable graph representation. This separation can cause inefficiencies (data copying between runtime environments) and complexity in verifying end-to-end behavior. Moreover, many solutions lack **real-time determinism**. For example, a ROS2 or DeepStream pipeline might achieve high throughput on average, but without careful design one can't guarantee microsecond-level jitter bounds – message queues and dynamic scheduling can introduce variability. Traditional LabVIEW on a PC wasn't designed for hard real-time control either – it often required a special real-time module or FPGA for deterministic timing. **Portability** is another issue: frameworks like TensorRT or Vitis are vendor-locked (NVIDIA-only, Xilinx-only), and even a "platform-neutral"

runtime like ORT doesn't inherently handle I/O or synchronizing with control loops – so developers resort to platform-specific code for those parts. In short, current SOTA tools tend to create *silos*: one for AI inference, one for control logic, one for hardware interfacing[46]. This makes it challenging to maintain a **single source of truth** for the entire AI system's behavior.

SOTA + ONNX GO HW: A Step Toward Unified, Real-Time Orchestration: *SOTA* (State Of The Art) and **ONNX GO HW** are recent initiatives (2025) by Graiphic that aim to address the above gaps by leveraging ONNX as a common graph representation beyond neural networks[47][40]. SOTA is a fully **ONNX-native AI framework** integrated into LabVIEW, which allows designing and even training deep learning models directly as ONNX graphs (with visual editing)[40]. More importantly for deployment, *ONNX GO* is a runtime that can **orchestrate ONNX graphs in real time** – essentially treating an ONNX graph as a program that runs across heterogeneous hardware, with ONNX Runtime under the hood for execution[41]. The upcoming **ONNX GO HW** extension goes further: it introduces standardized hardware-interfacing nodes (for DMA transfers, GPIO, ADC/DAC, timers, etc.) as part of the ONNX graph[48]. This means an engineer could represent **an entire system** – sensor input, decision logic (including AI model inference and classical code), and actuator output – as one ONNX computational graph. The ONNX Runtime then executes this graph end-to-end, calling out to hardware where needed (e.g. reading a sensor value into a tensor, feeding it through a neural net, then writing a control signal)[48]. All of it is scheduled by a single engine, rather than hopping between different runtimes.

Example: Unified graph orchestration on an SoC. In this architecture, a LabVIEW-designed diagram is compiled to an ONNX graph (containing AI models, control logic, and I/O nodes) and deployed to a target device. ONNX Runtime executes the graph as a data-plane on the device (utilizing hardware accelerators via execution providers, and performing I/O through hardware nodes), while a remote control-plane monitors and adjusts the system via secure RPC. This approach yields a single, portable graph representation governing the whole AI system, improving transparency and determinism.

By unifying everything in the ONNX graph, ONNX GO HW provides several advantages: **Full-System Graph Unification** – the entire pipeline is a single graph artifact, which can be inspected, tested, and versioned. This graph isn't just neural network layers; it can include conditional logic (ONNX *If* nodes), loops (*Loop*/*Scan*), and now hardware interactions, enabling complex scheduling and decision-making to be encoded declaratively[2]. Graiphic's CTO describes it aptly: "With [ONNX's] If, Loop, and Scan, a graph can decide, repeat, and orchestrate – not just what to compute, but when and how. Suddenly, an ONNX model is not frozen math; it's a living schedule"[2]. **Integrated Control and I/O** – ONNX GO HW's hardware nodes allow direct graph-level interfacing with devices[48]. For example, one could have an ONNX subgraph that reads a digital input, feeds it into a decision neural net, then through an *If* node decides whether to activate an output. Traditionally, the "glue" for such logic would be written in C++ or Python outside the model, but here it's part of the graph. This not only reduces development effort (no separate code for integration) but also ensures the entire execution can be analyzed for timing. **Portability** – ONNX graphs are portable by design; a single ONNX file can run on x86, ARM, NVIDIA GPUs, FPGAs (via Vitis), Qualcomm NPUs

(via QNN), etc., as long as there is an ONNX Runtime execution provider for that hardware. ONNX GO leverages this by allowing LabVIEW-designed systems to be deployed on "any hardware" without requiring the LabVIEW runtime on the target[49][50]. This is a significant shift: in one use case, they demonstrate designing a control system in LabVIEW and deploying it to run on a Raspberry Pi solely via an ONNX graph artifact[51]. In essence, the ONNX graph becomes a universal, **vendor-neutral bytecode** for AI systems. Finally, **Real-Time Determinism and Efficiency** – ONNX Runtime can compile a graph (especially with formats like ORT format or by using static execution planners) such that execution is *repeatable and time-predictable*. The LabVIEW+ONNX approach highlights determinism: *"ONNX Runtime [sessions] compiled once, [are] stable across runs"*, providing consistent timing[52]. Eliminating middleware layers (like a separate script invoking model inference) cuts down variability and latency – e.g., avoiding unnecessary buffer copies and context switches. Early indications (from Graiphic's demos) show ONNX GO can achieve millisecond-range response with low jitter for vision-and-control tasks at the edge[53]. The unified graph also simplifies **certification and validation** in safety-critical fields, since the whole logic (AI + non-AI) can be audited as one unit[54].

In summary, **ONNX GO HW** represents a convergence of ideas: it treats *everything* as a graph – not just neural nets, but loops, decisions, and hardware interactions – and uses a single runtime to execute that graph on any platform with high efficiency. Existing frameworks paved the way with optimized graph execution and modular pipelines, but they lacked this one-graph-to-rule-them-all unification. The combination of SOTA (visual design and training of ONNX models in LabVIEW) and ONNX GO HW (universal graph orchestration with hardware access) can be seen as a step forward toward **truly unified, portable, and real-time AI system orchestration**. It aims to deliver the ergonomics of LabVIEW, the portability of ONNX, and the performance of optimized runtimes, in one package[46][48]. If successful, this approach could significantly reduce the complexity of deploying advanced AI systems in domains like robotics, industrial control, and autonomous vehicles – empowering a single graph to reliably run an entire intelligent workflow from sensing to actuation, regardless of the underlying hardware.

Sources: The information in this review is drawn from recent scientific papers, industry documentation, and technology blogs. Key references include academic studies on neural network compilers and scheduling[18][4], surveys on graph neural networks[9], and documentation of frameworks like NVIDIA Triton[31], Holoscan GXF[33], ROS 2[38], DeepStream[36], Vitis AI[42], AMD ADF[43], and Qualcomm QNN SDK[25]. The discussion on ONNX GO HW and SOTA is based on reports from Graiphic's 2025 GLA Summit presentation[41][48] and follow-up articles[2][52], which outline this emerging unified graph approach.

[1] [18] [19] Leveraging Neural Graph Compilers in Machine Learning Research for Edge-Cloud Systems
https://arxiv.org/html/2504.20198v1
[2] [39] [46] [51] [52] [53] [54] LabVIEW Everywhere: From Gauges to Pilot - Graiphic
https://graiphic.io/labview-everywhere-onnx/
[3] Chainer: Dynamic vs Static Graphs - Tutorialspoint

https://www.tutorialspoint.com/chainer/chainer-dynamic-vs-static-graphs.htm

[4] [5] [21] usenix.org

https://www.usenix.org/system/files/osdi23-zhao.pdf

[6] A learnable dynamic scheduling for directed graph jobs flow in …

https://www.sciencedirect.com/science/article/abs/pii/S0167739X25003425

[7] [22] Research on computing task scheduling method for distributed …

https://www.nature.com/articles/s41598-025-94068-0

[8] Heterogeneous Graph Neural-Network-Based Scheduling … - MDPI

https://www.mdpi.com/2076-3417/15/10/5648

[9] [10] [12] [13] [14] [15] [16] [17] KDD 2023: Graph neural networks' new frontiers - Amazon Science

https://www.amazon.science/blog/kdd-2023-graph-neural-networks-new-frontiers

[11] Graph Neural Networks for temporal graphs: State of the art … - arXiv

https://arxiv.org/abs/2302.01018

[20] [PDF] An Automated End-to-End Optimizing Compiler for Deep Learning

https://www.usenix.org/system/files/osdi18-chen.pdf

[23] Comparing Task Graph Scheduling Algorithms: An Adversarial …

https://arxiv.org/html/2403.07120v1

[24] [26] ONNX | Home

https://onnx.ai/

[25] Qualcomm - QNN | onnxruntime

https://onnxruntime.ai/docs/execution-providers/QNN-ExecutionProvider.html

[27] Making AI Compute Accessible to All, Part 7: Inside the TVM Stack …

https://medium.com/the-software-frontier/making-ai-compute-accessible-to-all-part-7-inside-the-tvm-stack-and-its-lasting-impact-88f901788604

[28] Introducing TVM Auto-scheduler (a.k.a. Ansor)

https://tvm.apache.org/2021/03/03/intro-auto-scheduler

[29] Glow: Graph Lowering Compiler Techniques for Neural Network

https://medium.com/geekculture/glow-graph-lowering-compiler-techniques-for-neural-network-fb1eacbd0508

[30] eIQ® Inference with Glow NN - NXP Semiconductors

https://www.nxp.com/design/design-center/software/eiq-ai-development-environment/eiq-inference-with-glow-nn:eIQ-Glow

[31] [32] [45] Ensemble Models — NVIDIA Triton Inference Server

https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/ensemble_models.html

[33] [34] [35] Graph Execution Framework (GXF) - NVIDIA Docs

https://docs.nvidia.com/clara-holoscan/archive/clara-holoscan-0.3.0/gxf/index.html

[36] [37] DeepStream Components — DeepStream documentation

https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_Zero_Coding_DS_Components.html

[38] Understanding topics — ROS 2 Documentation: Foxy documentation

https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html

[40] [41] [47] [48] Graiphic Unveils SOTA and ONNX GO at GLA Summit 2025: Revolutionizing AI Development in LabVIEW - Graiphic

https://graiphic.io/gla-summit-2025-sota-labview-onnx-go/

[42] Exercising Vitis AI Applications on Alpha Data Boards V1.0
http://alpha-data.com/pdfs/ad-an-0131_v1_0.pdf
[43] [44] Developing a BLAS library for the AMD AI Engine Extended Abstract
https://arxiv.org/html/2410.00825v1
[49] Deploy without the LabVIEW Runtime, powered by ONNX GO HW
https://graiphic.io/labview-everywhere-onnx-go-hw/
[50] Edge AI Archives - Graiphic
https://graiphic.io/tag/edge-ai/