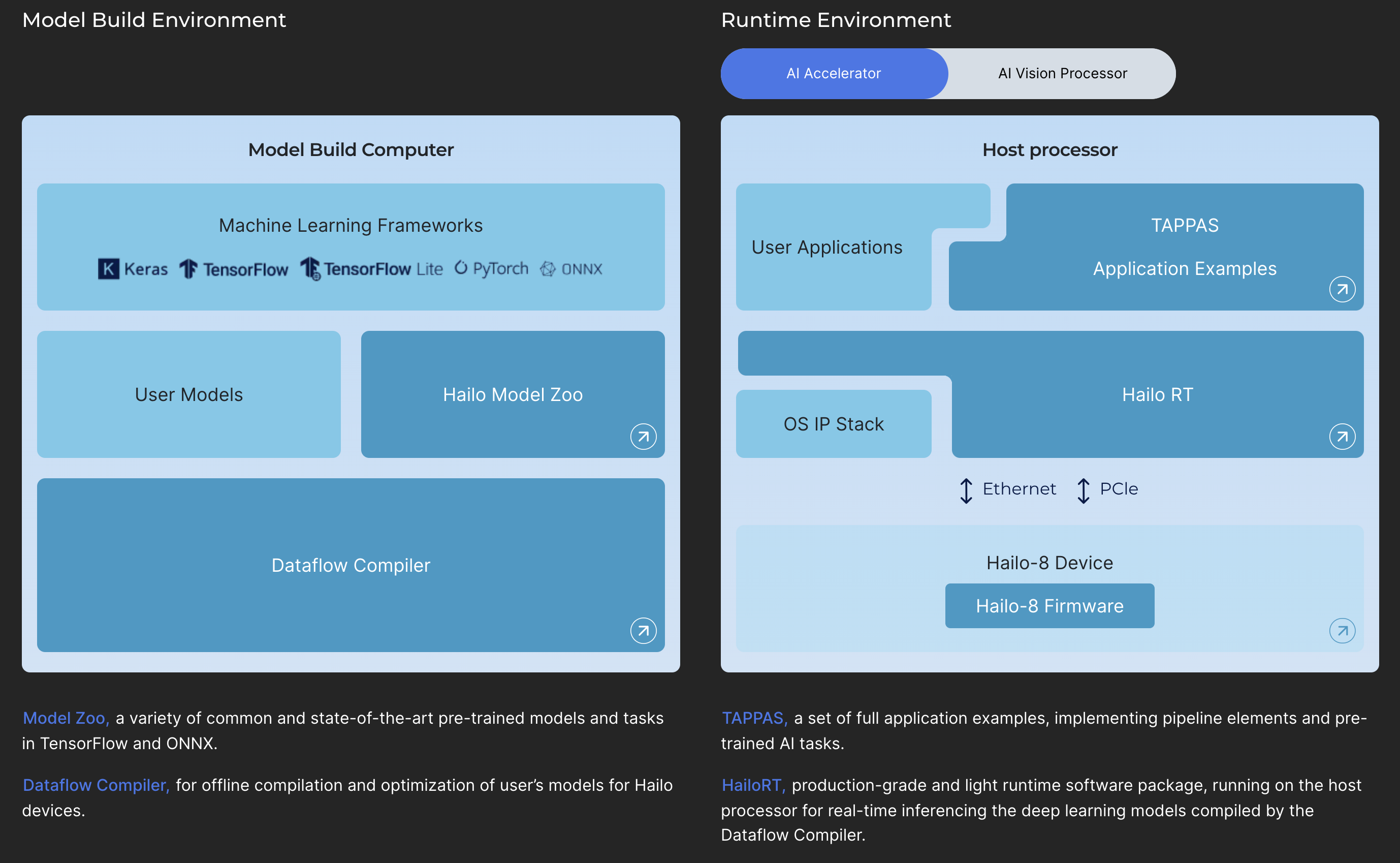
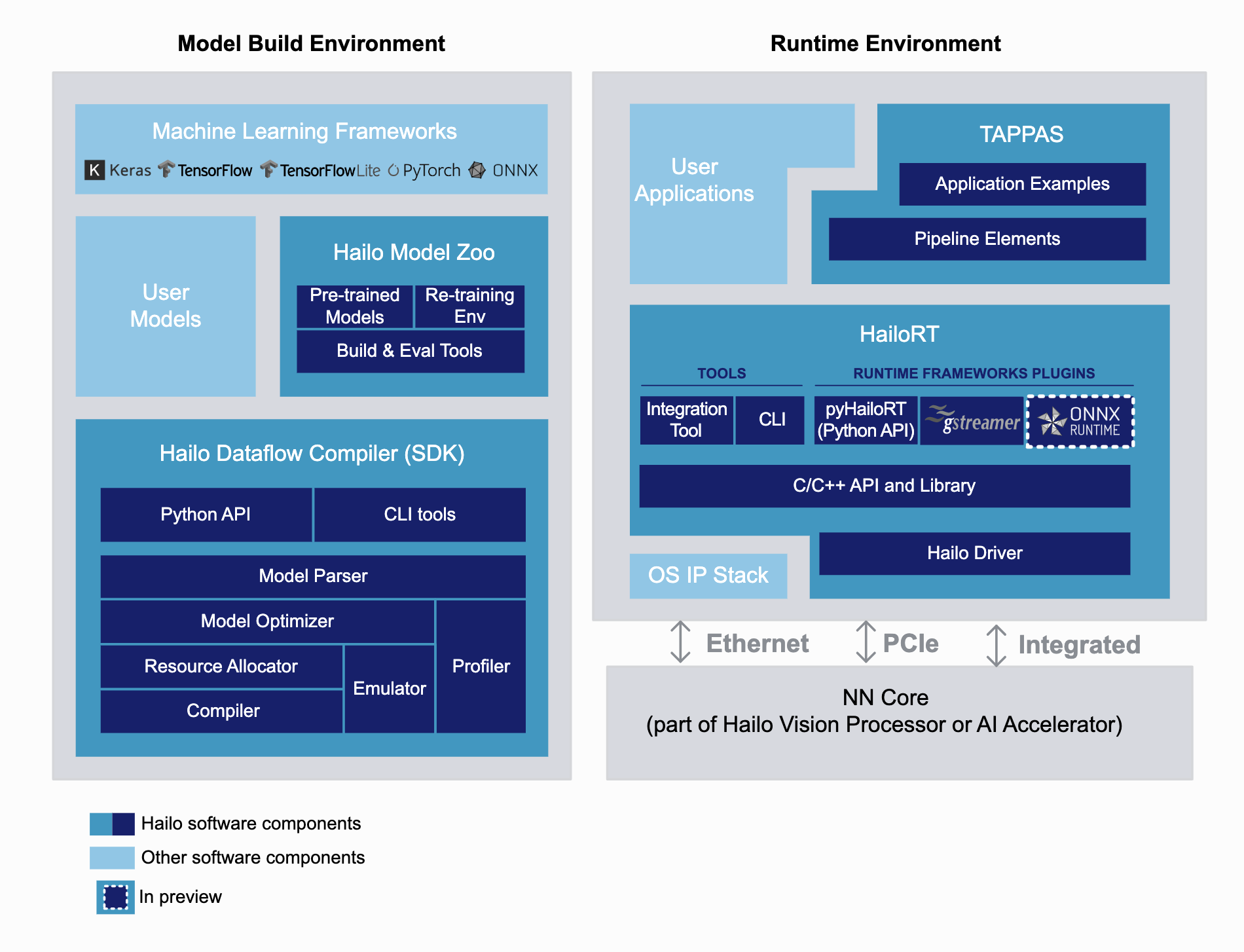
HAILO

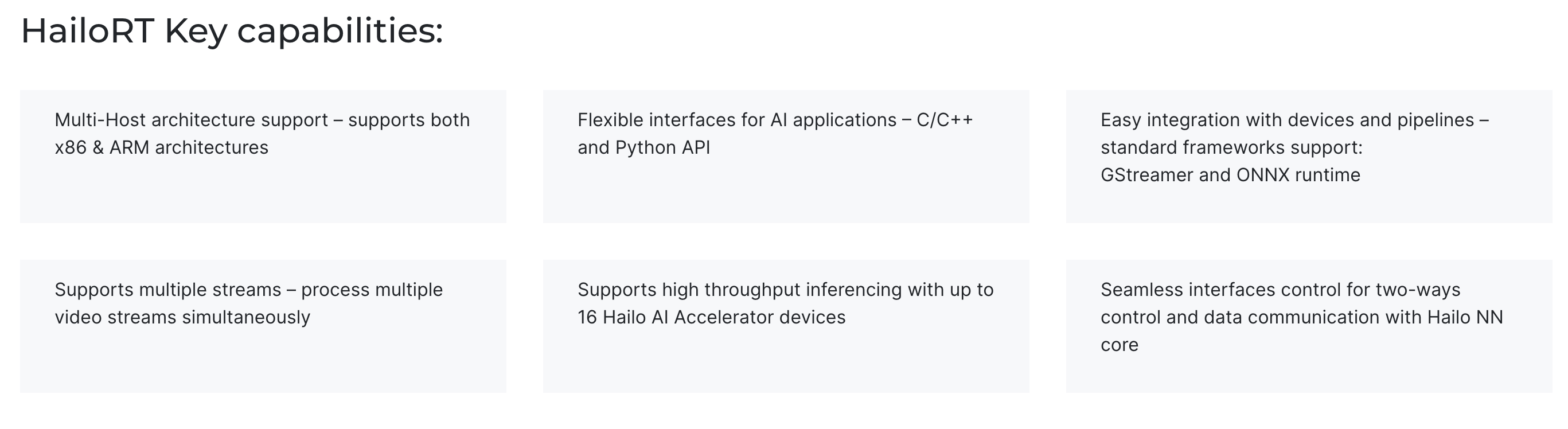
**Key components of HAILO Software suite**





# HAILO RT

Hailo SW products are set of frameworks and tools that enable you to compile, run and evaluate neural networks on Hailo devices.

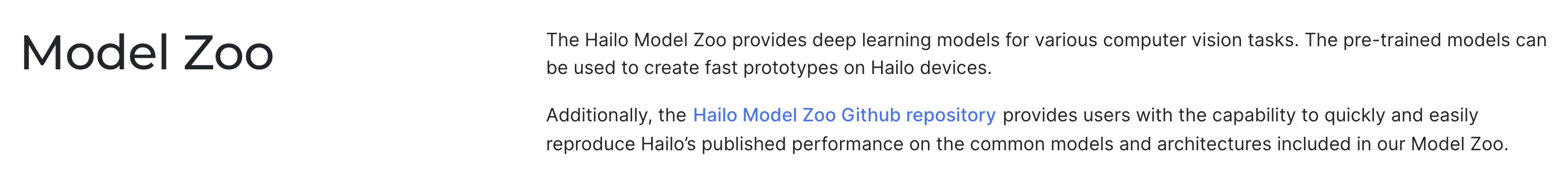


HailoRT consists of the following main components:

* HailoRT Library.
* HailoRT CLI - a command line application used to control the Hailo device, run inferences, collect statistics and device events, etc.
* [HailoRT PCIe Driver](https://github.com/hailo-ai/hailort-drivers) - the device driver used to manage the Hailo device, communicate with the device, and transfer data to/from the device; it includes the Hailo-8 firmware that runs on the Hailo device and manages its boot and control.
* pyHailoRT - HailoRT Python API, which wraps the runtime library.
* HailoRT GStreamer element (HailoNet).

HailoRT supports Linux and Windows, and it can be compiled from sources to be integrated with various x86 and ARM processors.

# HAILO Model Zoo



# Dataflow Compiler

Dataflow Compiler Tutorials Introduction

1. Converting a Tensorflow or ONNX neural-network graph into a Hailo-compatible representation.
2. Quantization of a full precision neural network model into an 8-bit model.
3. Compiling the network to binary files (HEF), for running on the Hailo device.

**Parsing (Translating)**

* Hailo Parsing Example from Tensorflow CKPT to HAR.
* HAR is a tar.gz archive file that contains the representation of the graph structure and the weights that are deployed to the Hailo hardware (like Relay in TVM).
* Does support Quantization Aware Training.
* Parsing Example from ONNX to HAR.

**Model Optimization**

HAR model file => HAR Quantized model files.

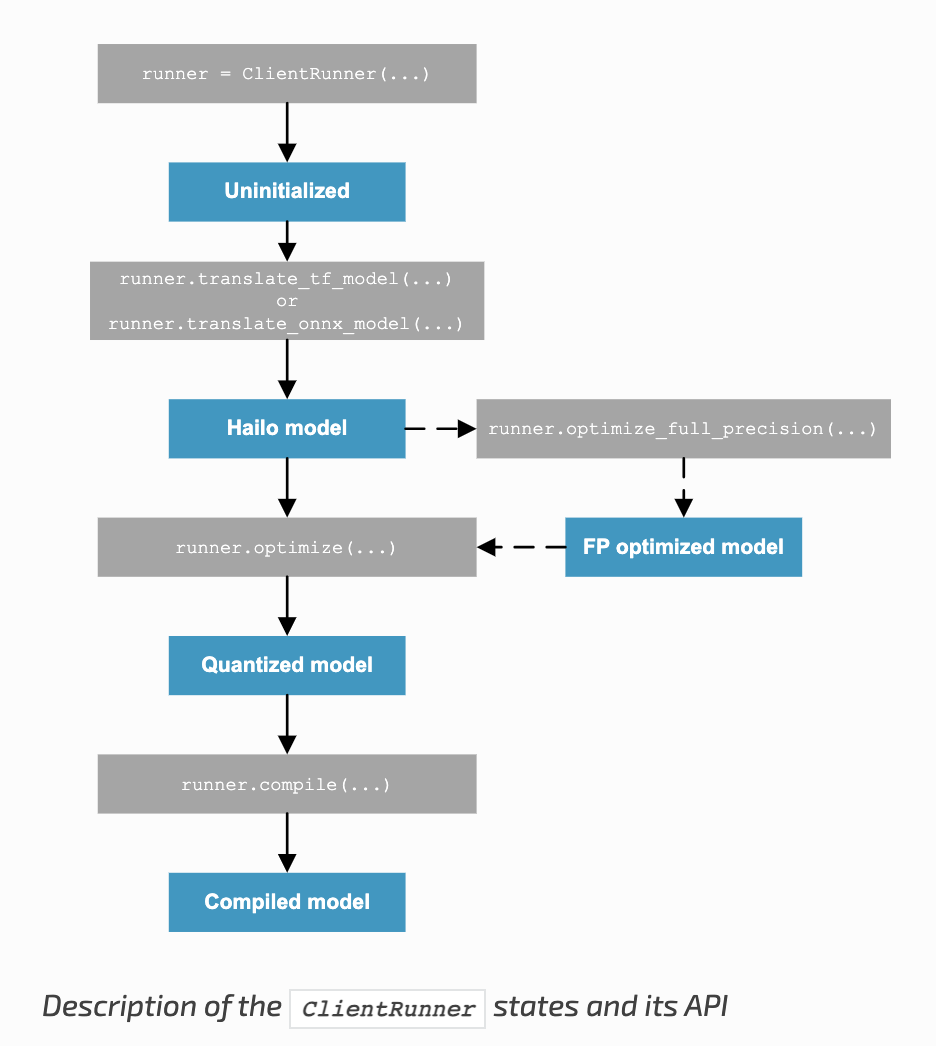
This tutorial describe the process of optimizing the user’s model. The input to this tutorial is a HAR file in Hailo Model state (before optimization; with native weights) and the output will be a quantized HAR file with quantized weights.

**Compilation**

Quantized HAR model file => Hailo8 binary files (HEF).

**Inference**

Inference on Hailo devices



[Supported layers](https://hailo.ai/developer-zone/documentation/dataflow-compiler-v3-26-0/?sp_referrer=sdk/supported_layers.html)

TVM

The reason I find this interesting is as follows:

* BYOC has been a successful interface between TVM and vendors
* VTA is the historical template of accelerator programming with full TVM stack
* Yet the VTA codebase does not, currently, have a BYOC adaptation

# VTA

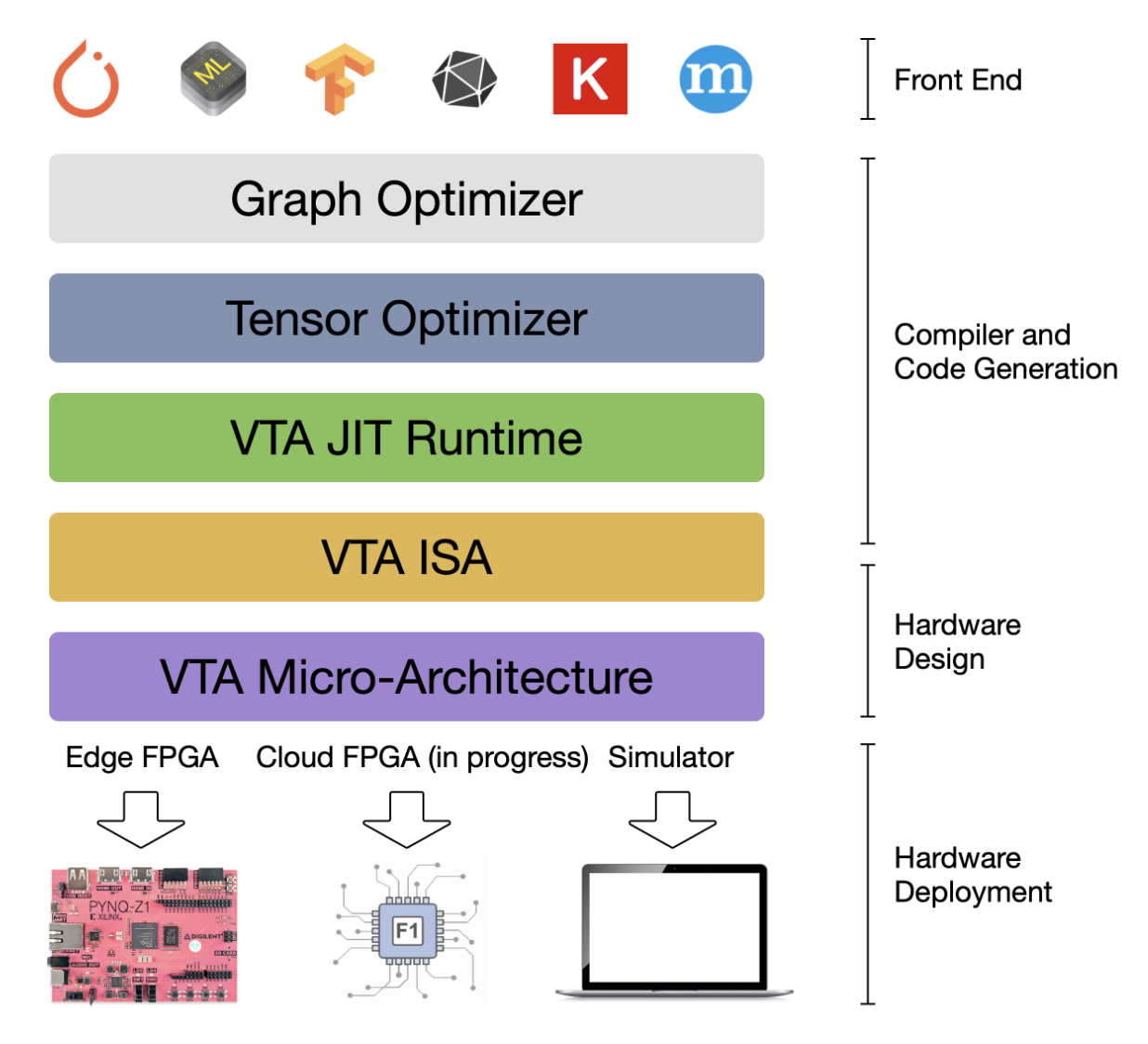
(CUSTOMABLE DEEP LEARNING ACCELERATOR)

VTA (versatile tensor accelerator) is an open-source deep learning accelerator complemented with an end-to-end TVM-based compiler stack.

The key features of VTA include:

* Generic, modular, open-source hardware
  + Streamlined workflow to deploy to FPGAs.
  + Simulator support to prototype compilation passes on regular workstations.
* Driver and JIT runtime for both simulator and FPGA hardware back-end.
* End-to-end TVM stack integration
  + Direct optimization and deployment of models from deep learning frameworks via TVM.
  + Customized and extensible TVM compiler back-end.
  + Flexible RPC support to ease deployment, and program FPGAs with the convenience of Python.

Learn more about VTA [here](https://tvm.apache.org/docs/vta/index.html).



# Technical Detail

**NNVM**

The graph-level optimizer, provides a graph-level Intermediate Representation (IR) used as a common language between different deep learning frameworks to take advantage of graph-level optimizations, such as operator fusion.

The NNVM IR is also used to specify data layout and data format constraints: e.g. tiling for tensorization, and bit-packing for ultra-low precision computing.

**TVM**

The tensor-level optimizer, builds upon the Halide DSL and schedule primitives to provide an optimizing compiler capable of bringing performance portability for deep learning across hardware back-ends.

TVM brings novel scheduling primitives that target specialized hardware accelerators, such as tensorization, which lowers computation onto specialized tensor-tensor hardware instructions. In addition, it provides schedule primitives and lowering rules that allow for explicit memory management to maximize resource utilization in hardware accelerators.

**VTA**

The VTA runtime performs JIT compilation of VTA binaries (instruction streams and micro-kernel code), manages shared memory, and performs synchronization to hand off execution to VTA. The VTA runtime presents an API that looks generic to TVM, to hide complexities of platform-specific bookkeeping tasks. It exposes a C++ API that a TVM module can call into - this simplifies the future inclusion of other hardware accelerator designs, without having to drastically modify the upper TVM layers.

VTA’s two-level ISA provides both

1. a high-level CISC ISA that describes variable latency operations such as DMA loads, or deep learning operators and
2. a low-level, and fixed latency RISC ISA that describe low-level matrix-matrix operations. This two-level ISA allows both code compactness, and expressiveness.

Finally, VTA’s micro-architecture provides a flexible deep learning hardware design specification, that can be conveniently compiled onto other FPGA platforms, and eventually in the long term down to ASICs.

------

Relationship between VTA and BYOC

https://discuss.tvm.apache.org/t/byoc-and-the-vta-missing-link/6867/2

TVM UMA

https://github.com/apache/tvm-rfcs/blob/main/rfcs/0060\_UMA\_Unified\_Modular\_Accelerator\_Interface.md

The goal of **UMA (Universal Modular Accelerator Interface)** is to create a unified infrastructure for easily integrating external accelerators into TVM. UMA provides file structures, Python interface classes and an API for accelerator integration. These interfaces and API are accessible from Python and are part of the components *UMA Partitioner*, *UMA Lower* and *UMA Codgen*. The features and proposals of *Target registered compiler flow customization* [TVM-RFC0011] and [TVM-RFC0010] are considered, with the difference that UMA tries to provide a more general interface for integrating new accelerators and one specific implementation of the hooks described in [TVM-RFC0011].

TVM port to custom accelerator

https://discuss.tvm.apache.org/t/feedback-on-tvm-port-to-custom-accelerator/9548

BYOC to TVM

https://tvm.apache.org/2020/07/15/how-to-bring-your-own-codegen-to-tvm