EC7 AS15

User Documentation of the Confiance.ai ML performance evaluation bench

Document reference number for ANR

contact@confiance-ai.fr | www.confiance.ai

Document reference: 715A

Contributors

	Name	Organisation	Role
Responsible for the deliverable	E. JENN	IRT Saint Exupery	AS lead
Scientific responsible	E. JENN	IRT Saint Exupery	AS lead
Co-authors	F. THIANT E. JENN S. TEMBO-MOUAFO	IRT System-X IRT Saint Exupery IRT Saint Exupery	Authors

Document control

Revision	Date	Commentary	Author
1.0	2023/12/15	1st version delivery	See authors

Table of contents

A. Introduction and Abstract 4

A.1 General introduction to trustworthy AI challenges 4
A.2 Specific challenges 4
A.3 Objectives of the document 5

B. Global description 5

B1. Objectives of the benchmarking environment 5
B.2 Context 6
B.3 A typical use case 7
B.4 Architecture overview 8

C. Software architecture 9

D. Hardware architecture 17

D.1 The bay and racks 17
D.2 The Monitoring boards 18
D.3 The managed USB boards 19

E. Status and further developments 22

F. Bibliography 23

A. Introduction and Abstract

A.1 General introduction to trustworthy AI challenges

Trustworthiness in AI within critical systems (systems that can directly or indirectly affect human life and moral entities) is essential for its widespread adoption (by the industry, the decision makers, the general public, etc.) and poses the following significant challenges.

- First, how to design AI models, so that, by construction, they satisfy trustworthy properties (accuracy, robustness…).
- Secondly, how to characterize these AI models, for example to understand and explain their behavior and their adequacy to the operational domain.
- Then, how to implement and embed those AI models on hardware, by making them fit for the target without losing their trustworthy properties.
- Another question is, what methods of data engineering to apply in order to, among other topics, manage important volumes of data and adapt to the evolution of the operational domain.
- At system level, what verification and certification processes to consider specifically for AI-based systems.
- Finally, a federation of all these matters is necessary to build an end-to-end methodological approach, supported by a consistent engineering environment compatible with industrial practices.

These are the challenges, among others, that the Confiance.ai program addresses.

A.2 Specific challenges

The benchmarking environment specifically addresses the third challenge concerning the implementation and deployment of an AI model. Indeed, once the AI model has been designed and trained, and functional performances have been demonstrated to comply with the requirements, the model needs to be deployed and executed on some hardware using a series a transformation including (possibly) platform-specific optimizations, translation to specific forms, compilation, etc. This process raises multiple new challenges. First, it is highly platform-dependent, and different targets usually require – at least partially – specific tools. Second, those tools are often pretty complicated to setup, sometimes due to some lack of maturity, sometimes due to complicated dependencies between libraries, or simply because they require specific skills (e.g., FPGA development) or knowledge (e.g., specific library). Third, comparing performances and resource usage for multiple hardware targets requires to buy and setup the hardware targets, which is expensive in time and money. The benchmarking environment tries to address all those challenges by providing a set of “encapsulated” AI development tools and hardware targets (i.e., computation boards) ready to be used with a minimum effort. The environment also provides the necessary scripting, hardware management, and orchestration means to automate a series of experiments.

A.3 Objectives of the document

This document gives an overview of the architecture of the ML performance evaluation bench.

B. Global description

B1. Objectives of the benchmarking environment

The aim of the evaluation testbench is to facilitate the evaluation of implementation and deployment solutions of ML models on embedded platforms. It provides an easy access to a plurality of hardware targets including CPUs, GPUs, FPGAs, hardware accelerators and associated proprietary and open-source tool chains.

The platform is designed to simplify access to ML implementation and deployments tools in an ever-changing technical landscape (framework, hardware), and to be integrated into the development flow of an ML engineer/researcher wishing to minimize model embeddability failures as early as possible :

- Upstream :
  - Guide the user in the choice of frameworks/hardware targets best suited to his problem, as well as an experimentation methodology
  - Provide the user with initial performance indicators for his problem, for similar problems (database of experiments), or even for architectures not supported by prediction models
- Downstream :
  - Enable rapid prototyping of a controlled and constrained deployment pipeline
  - Build and enrich a reference knowledge base on ML model performance
  - Offer an up-to-date compression brick

The main capabilities of the benchmarking environment are depicted on Figure 1.

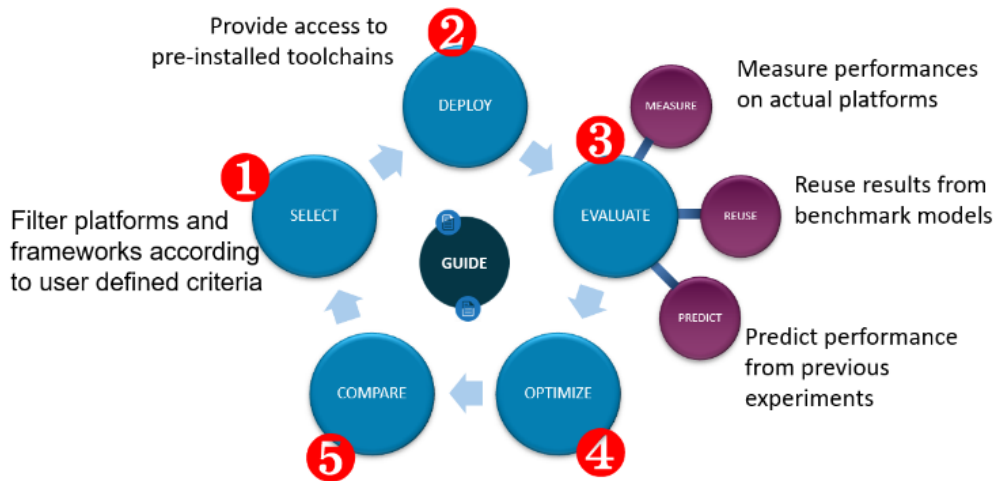

Figure 1. Main capabilities of the benchmarking environment

B.2 Context

The exploration of machine learning models on resource-constrained devices, commonly referred to as **TinyML**, represents a convergence of various technological domains. This includes hardware platforms with ample power to support standard operating systems like Linux. Within this realm, operational challenges emerge, introducing a need for innovative solutions to address the constraints imposed by limited computational resources, memory, and energy on these devices.

One critical aspect of the computational challenge involves employing compression techniques to curtail the computational cost, memory footprint, and energy consumption associated with running machine learning models on resource-constrained devices. The crux lies in establishing a strategic pairing between the implementation chain and the hardware target, as this synergy is essential to mitigate performance degradation.

Moreover, the environment in the TinyML sub-domain is notably fragmented, marked by the coexistence of various machine learning frameworks and heterogeneous hardware architectures. This fragmentation adds an additional layer of complexity to the development and deployment processes, necessitating adaptability and interoperability across diverse platforms, to cover this overlapping of technological domains as illustrated on Figure 2.

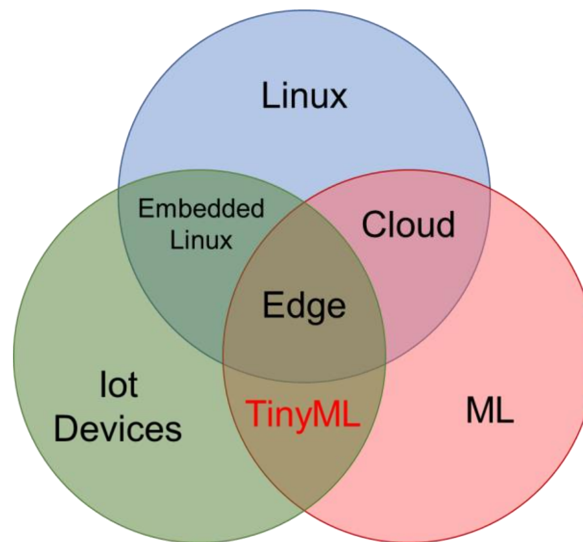

Figure 2. TinyML technological domains

To tackle the challenges of TinyML, **TinyMLOps** enters the scene, bridging the principles of MLOps, a set of practices that streamline the end-to-end lifecycle of machine learning models. From development and training to deployment and monitoring, MLOps aims to enhance collaboration, automate workflows, and ensure the reliability of machine learning systems (in our context, we do not consider the training part).

Extending these MLOps principles to the realm of TinyML must cover the following operational challenges :

- Computational aspect: use of compression techniques to reduce computational cost, memory footprint and energy consumption.
- Highly fragmented space, with various machine learning frameworks coexisting alongside heterogeneous hardware architectures, complicating the development and deployment processes

B.3 A typical use case

The user wants to deploy a model on hardware that meets the following criteria:

- Inference precision and latency compliant with requirements (requires actual inference on the target and its measurement).
- Hardware component at a fair cost (requires adding cost elements).
- The most transparent implementation chain possible (requires leveraging the analysis documents we are supposed to produce).

Here is a typical interaction scenario with the testbench:

1) The user selects a model to deploy.
2) The user provides constraints on the targets and tools (e.g., using European or U.S. hardware targets, not using the TensorFlow2 framework, etc.). In the first version of the tool this will be achieved using requests to the benchmarking environment.
3) The tool proposes a list of configurations compatible with this model and the constraints given by the user. Compatibility is defined by (i) support for operators used by the model, (ii) compatibility with resources available on the target (available memory), (iii) estimation of the model's temporal performance.
4) The tool provides direct access to some high-level documentation regarding (i) the hardware target and (ii) the tools supported by the targets. It also provides some guidance collected by users (including the testbench developer) about the tools.
5) After consulting the documentation, the user decides to choose a GPU target that requires no modification to their model (for example, Jetson Orin).
6) The user selects the workflows to be executed. In this scenario, the user first wants to evaluate performance measurement on a Jetson Orin target.
7) The tool executes the workflows on the Jetson Orin
8) The user retrieves and analyses the results.
9) The user wants to know the time it would take for inference on smaller Jetson targets (e.g., Jetson Nano or Xavier). Unfortunately, the two targets are not available (experiments are underway on those targets, so the user selects the corresponding “predicted targets” that correspond to using the performance prediction model.
10) The user now wants to estimate performances for other configurations using a CPU target among those proposed in the bench (e.g., (Raspberry Pi,TFLite), (TDAV4M, TIDL), etc.). This is achieved using a Python script that loops over the list of possible configurations generated by the tool.
11) The user selects a Xilinx FPGA target.
12) The tool proposes two workflows: one using the VitisAI toolchain and another using the finn toolchain.
13) From all the collected results, the user selects the target with appropriate ML and timing performances and the lowest memory footprint.

B.4 Architecture overview

Following the TinyMLOps principles, we implemented an architecture which reflects the overlapping of technological domains of TinyML, as illustrated on Figure 3, and reported on the architecture overview on Figure 2, with the same color legends for the domain circles (**ML**, **Embedded**, **Linux**).

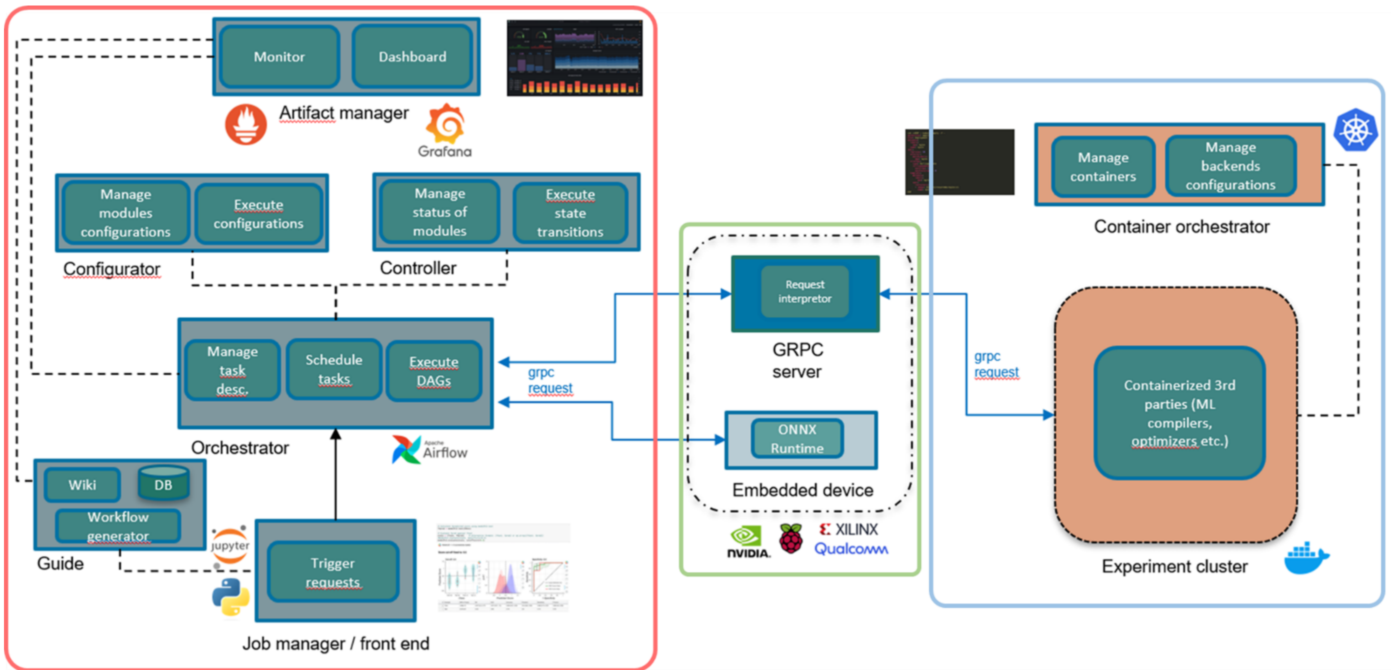

Figure 3. Architecture overview

C. Software architecture

A micro-services architecture

The evaluation bench is composed of a set of components encapsulated in [Docker] containers providing services accessed through [gRPC], a RPC (Remote Procedure Call) framework based on HTTP/2.

The set of services provided by the bench covers in particular the management of the ML experiments lifecycle, i.e., set up experiments, implement and deploy model, run i.e., power-up/down boards, manage health status, monitor temperatures, etc.

Those services are accessible via a Python API that provides the user with a uniform interface despite the heterogeneity of the set of execution platforms that includes both high-performance targets running Linux, Android, and others with no operating system.

In order to leverage existing standards for ML deployment, we integrate the [Nvidia Triton Inference Server], an opensource inference serving solution optimized for both CPUs and GPU, which implements the [Predict Protocol], (a predict/inference API independent of any specific ML/DL (DeepLearning) framework and model server. Extension of this inference server is planned to handle unsupported hardware or execution implementation, also called backends (e.g.: PyTorch, TensorFlow, TensorRT, etc.). In a way similar to the PredictProtocol, which is in fact a gRPC schema based on ProtocolBuffers, a data serializing protocol, an independent API is provided for each class of services necessary to automate the ML pipeline

(e.g.: Deployment Protocol, Conversion Protocol, Quantization Protocol etc.), but without being restricted to one type of hardware and corresponding software tools.

- Example of Predict Protocol grpc request :

```
channel = grpc.insecure_channel(endpoint_addr: port')
# create a gRPC stub
stub = prediction_pb2_grpc.ModelStub(channel)
# create a gRPC request
request = prediction_pb2.PredictRequest(input=input_data)

# cake the gRPC call
response = stub.Predict(request)
```

The experiment workflow

A typical experiment corresponds to the sequence presented in Figure 4.

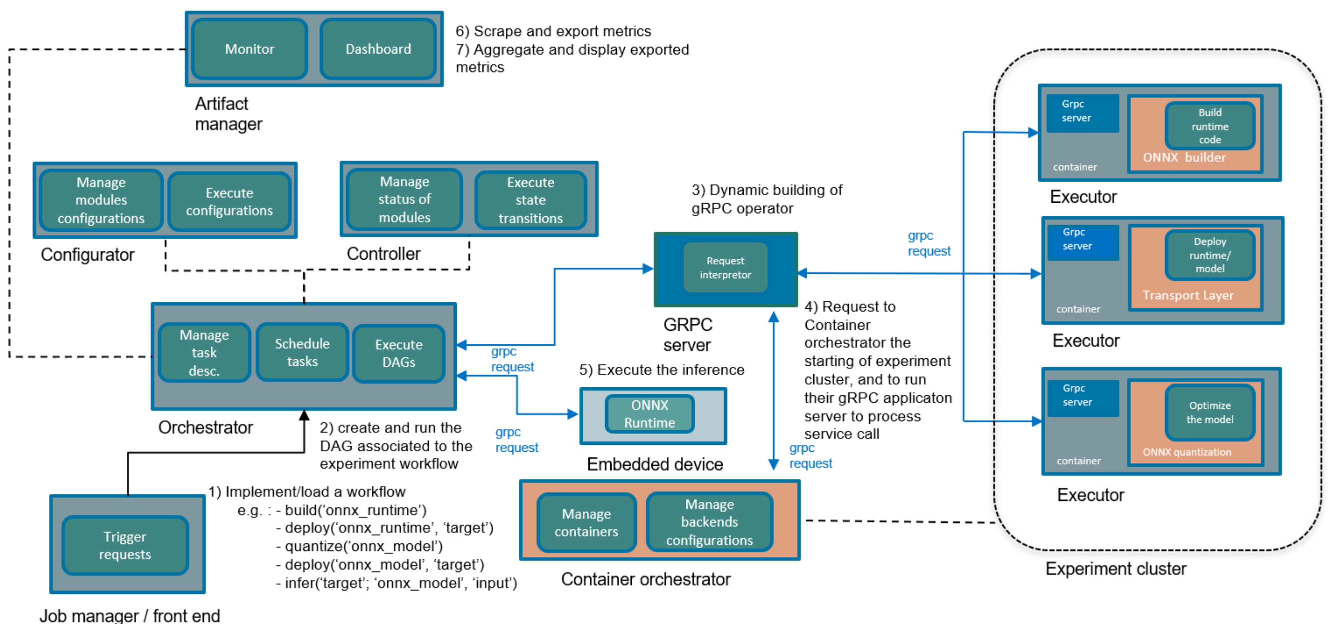

Figure 4. Execution overview with ONNX environment

- Front-end (Python API / JupyterNotebook):
  - the user can implement its own workflow or load a predefined optimized workflow, i.e. a sequence of couples (service/ component), also called an Executor, which is a docker container implementing its own grpc server able to process the type of service desired (e.g. *serviceInference(model, shape, input)*). This sequencing results in the creation of a dynamic Airflow DAG (Directed Acyclic Graph).

- Orchestrator (Airflow)
  - the Configuration module provides all the parameters required for the launching of components, as well as the parameters required by the services, i.e. the remote functions to call the Controller module is a state machine using Airflow cross-communications, applied at services level (*init/config/start/stop/deinit*)

- Request Intepretor (gRPC)
  - an Airflow gRPC operator, i.e. the task to perform, is built based on a service serialization/deserialization schema (also called a plugin)

- Executors (Docker)
  - call the local function relative to the building/deployment of the experiment (model conversion/quantization, toolchain building blocks, etc.). All the containers composing an experiment are grouped within a cluster, thus providing isolation, and managed at by the container orchestrator (Kubernetes, but always under control of the main Orchestrator)

- Hardware device
  - call the local function relative to inference, or to the board monitoring

- Monitoring (Prometheus)
  - collect and store the experiment (hardware/ML metrics) and monitor the platform (system health, behavior, and performance), using Prometheus HTTP endpoint or raw text files for low-end target

- Visualization (Grafana)
  - dashboard of the entire experiment

The experiment configuration

Aged by a dedicated component, the Configurator, which reads yaml files and supplies all components with the configuration parameters they require, enable dual operation: function calls with parameters, in line with grpc client-server architecture, or, for greater flexibility, direct parameter retrieval by a developer wishing to quickly integrate a new toolchain, without necessarily having to implement all the necessary services.

In addition, this component, responsible for storing configuration data in a database, also holds a set of global variables representing the state of the system (e.g. : *flag_run == result_of(is_an experiment_running , board_1), flag_available == is_available(board 2)*) avoiding over-querying, therefore minimizing interference/sources of variability on hardware behavior.

The configuration files ingested by the Configurator are in yaml format, as illustrated on Figure 4, representing the environment to build and deploy, the target to reach, the ml application to run, the measurement to perform, or the full automatized workflow to execute.

As mentioned in the previous section, the user-generated DAG (also called ExpDAG) interacts with the Configurator to retrieve the parameters needed to build the Airflow task to be executed, i.e. a grpc task ([Airflow Grpc Operator](#)). The building of user request, and the full process leading to the generation of the new grpc request is illustrated on Figure 5.

This DAG, which generically applies a sequence of controllable transitions via the application of a schema (*init, config, start, stop, deinit*), will eventually be integrated into the Configurator to generate as much as possible of the implementation enabling the DAG to be executed under constraints, i.e. the preceding schema or extended according to the desired granularity (e.g. : for all optimization services of type quantization, apply the following control schema : *call_static_quantization -> call_recalibrate*).

```yaml
config_files:
  - !include config_services_k8s.yml
  - !include config_envs.yml
  - !include config_workflows_k8s.yml
  - !include config_targets.yml
  - !include config_components.yml

applications:
  - application: !application
      identifier: app_1
      model_name: densenet_onnx
      model_folder: $MODELS_REPO
      data_name: mug.jpg
      data_folder: $MLTB_DATASET/images
      scaling : INCEPTION,
      shape : [3, 224, 224]
      dformat : FORMAT_NCHW
      dtype : FP32

measurements: # i.e metrics
  - measurement: !measurement
      identifier: measure_1
      mname: nv_inference_compute_infer_duration_us

optimizations:
  - optimization: !optimization
      identifier: optimization_1
      otype: quantization
      options: [dynamic, int8]

exps:
  - exp: !exp
      identifier: exp_1
      name: my_exp1
      application: app_1
      workflow: workflow_1
      target: testbench_1
      measurements: [compute_infer]
      optimizations: [optimization_1]
```

Figure 5. Bench configuration file

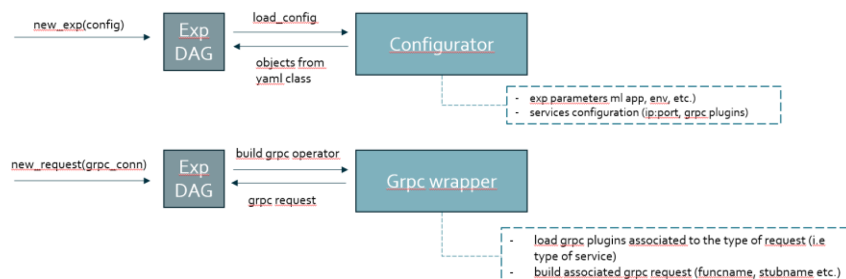

Figure 6. User request processing

The experiment cluster

The services of the benchmarking environment are implemented by components packaged in the form of Docker containers. To facilitate their management, and in keeping with a TinyMLOps approach, the container orchestrator Kubernetes was chosen, in particular because of the specific nature of our platform, which has to interact with hardware connected to physical servers located in different geographical locations.

According to the Kubernetes terminology, these components/dockers are called Pods, a group of one or more containers with shared storage and network resources, and a specification for how to run the containers. All the Pods required to perform an experience, i.e. to execute the actual services using grpc calls, are inter-connected via a logical sub-network to mask the complexity of network addressing. For this purpose, we have superimposed the concept of services to Kubernetes Service, a method for exposing a network application that is running as one or more Pods in the experiment cluster. In our case, this means exposing the grpc server running in the pod, whose ip address provided by the bench configuration file is translated into a Kubernetes Service name.

Because of the heterogeneity of hardware targets, and therefore the communication protocols supported (serial, TCPI/IP etc.), a dedicated Pod, a containerized proxy server, has been implemented to handle them. This proxy server, the only Pod authorized to access to host network, is capable to communicate with remote containers (for high-level targets which run Linux); this solution has been implemented because of Kubernetes overhead, even for lightweight alternatives.

Figure 7 illustrates the role of the pods for the execution of the workflow described in Figure 4, at cluster level.

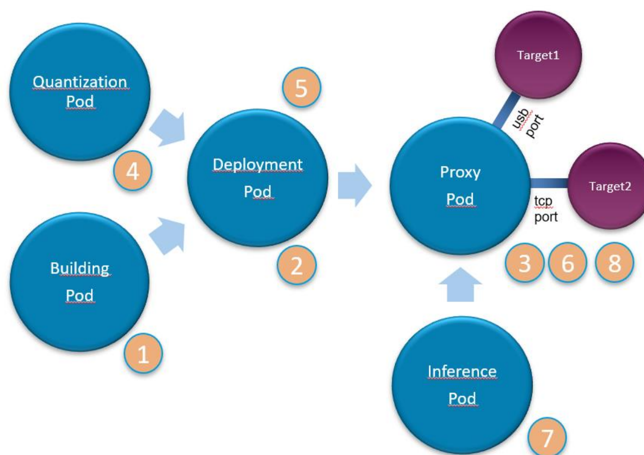

Figure 7. Execution workflow and pod interaction

The performance predictor

This performance predictor component is described in detail in a dedicated deliverable. It provides the user with the capability to estimate the performance of a (model, board) combination using a predictive model.

This feature allows the user to get a rough but fast estimation of performances without resorting to a complete implementation, deployment and inference workflow. In addition, it also allows users to obtain estimations when the actual physical boards are used (see architecture on Figure 8).

As of December 2023, this component supports performance prediction for models deployed on the Jetson Nano and Jetson Orin boards. In practice, there is one prediction model per board that is trained on a set of real and synthetic inference models. Future work will first extend this capability to CPU boards (ARM Cortex A78 family). In a second phase, we will investigate the capability to predict performances on a set of boards with different hardware characteristics.

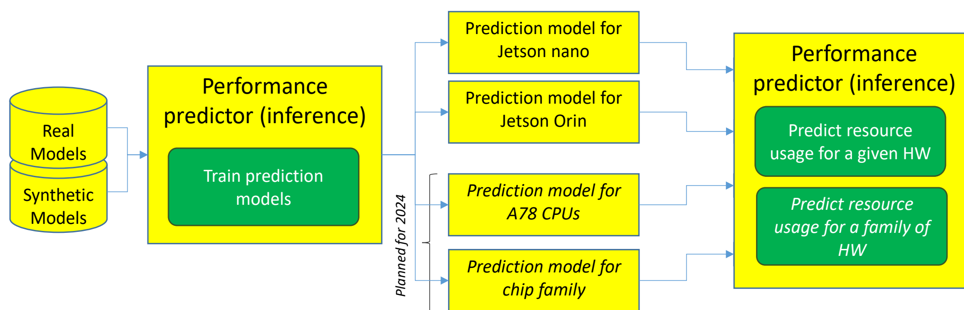

Figure 8. The performance predictor architecture

The workflow generator

The bench environment is fitted with a component that supports the generation of deployment workflows according to a set of user constraints.
The workflow generator has 7 components including 2 user front-end components and 5 back-end components. Front-end components directly interact with the user. The user does not interact with backend components.

1. User front-end components
   a. Jupiter notebook
      It is the user interface from which he selects some constraints inside a set of available workflow related constraints. At this step, constraints should be intelligibly displayed such that any user can understand them. Note that these constraints selected from Jupiter notebook interface are software constraints and performance constraints.

Software constraints relate to framework, converter or converter chain, embedded framework, delegate, backend and other tools. Performance constraints relate to metric such that accuracy, latency, memory consumption etc. Note that this Jupiter notebook interface is not yet implemented. It is part to future works related to the integration of the workflow generator on the mltestbench infrastructure.

b. MyQSL Database

The user can also specify hardware constraints, using SQL queries to the bench database, which contains structured information about boards, SoMs, SocC, targets, accelerators, manufacturers, etc.

2. Back-end components

Among the 5 back-end components, there are 3 lightweight components (parser and serializer scripts of user constraints) and 2 core components (Python API and Prolog engine). For now, only core components are already designed and implemented.

c. Parser/serializer scripts

The function of each of these scripts is to simply read human intelligible constraints and format them into a suitable form, e.g. a key-value structure, and serialize this dictionary into a constraint specification file, e.g. a yaml file.

d. Python API

It is a script with 2 parameters: the input constraint specification file created by parser/serializer scripts and the output file of generated workflows. For now, the constraint specification file is manually created. Python API performs the following tasks: 1- reads the constraint specification file, 2- generates on the fly prolog predicates suitable to specified constraints and queries the prolog engine, 3- waits until prolog results (raw workflows generated by the Prolog engine) are available, 4- retrieves and processes these workflows, 5- serializes these workflows into a workflow specification file, e.g, a yaml file.

e. Prolog workflow generator engine

It is a set of prolog predicates. A predicate is a fact, which can be conditionally or unconditionally true. A goal is a predicate or a combination of predicates that we want to run. The prolog runtime evaluates a goal by running a process called unification, i.e., searching all facts for which the goal is true. This research is more often performed recursively until the runtime does not find any more fact for which the goal is true. Indeed, a large number of core predicates of the Prolog engine perform recursively. This is because Prolog is a language with no built-in loop instructions such as for, while, etc. However, recursiveness replaces these loop instructions. The developer should be careful about infinite recursion by rigorously defining the knowledge base with facts and rules.

For example, to evaluate that a list of workflow activities is a valid sequence, we write a predicate called is_valid_sequence(list) such that it is true if: starting by the list head activity, each activity and the next are a valid sequence, and a unique activity is considered as a valid sequence in order make available to the prolog runtime a fact which allow it to stop the recursivity.

The Prolog workflow generator engine receives goals related to workflow generation from Python API, searches all workflows for which the received goal is true, and returns these generated raw

workflows to the python API. We say that these workflows are raw because each of them is a multi-level list not easy to read by a human. Python API tackles this problem by formatting a raw workflow into an easy-to-read activity sequences.

The Wiki

The bench hosts a Mediawiki server that provides the end users with a first level of documentation about the toolchains and hardware boards installed in the bench. Part of the documentation is generated automatically from the local database of cores, SoCs, boards, and tool chains.

An overview of the Wiki architecture is given Figure 9. A snapshot of the main page is given on Figure 10.

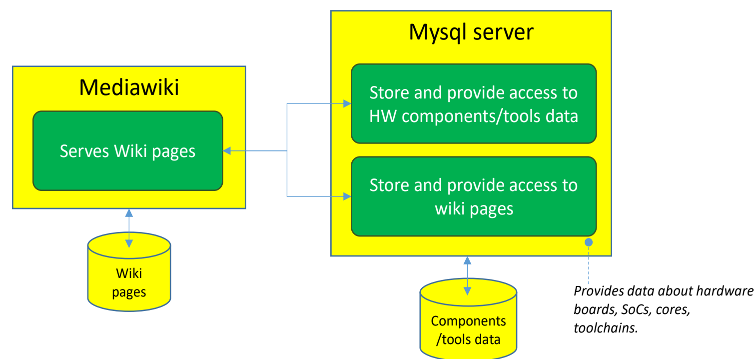

Figure 9. The Wiki architecture

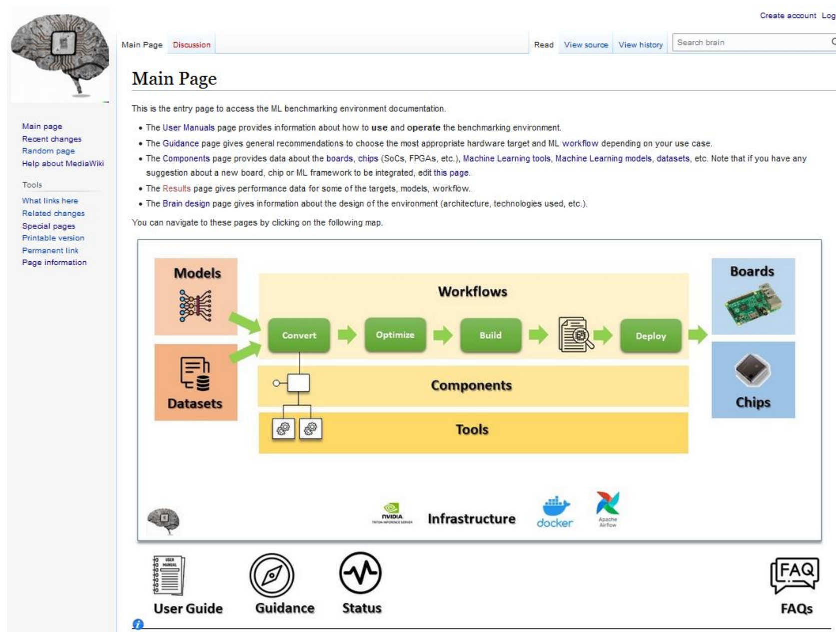

Figure 10. The Bench Wiki

D. Hardware architecture

D.1 The bay and racks

The components of the testbench, including the computer and its display, the targets boards and their power-supplies, the monitoring boards, the USB managed-switches, the ethernet switches, etc. are hosted in a unique bay whose picture is given in Figure 11, (a). The bay hosts a set of racks (b) and each rack is fitted with a set of target boards (c). Each rack is fitted with a dedicated “monitoring board” whose role is to drive the main power supply and provide information and support interaction with the user via a small front panel.

The main computer is connected to the internet through a secured connection.

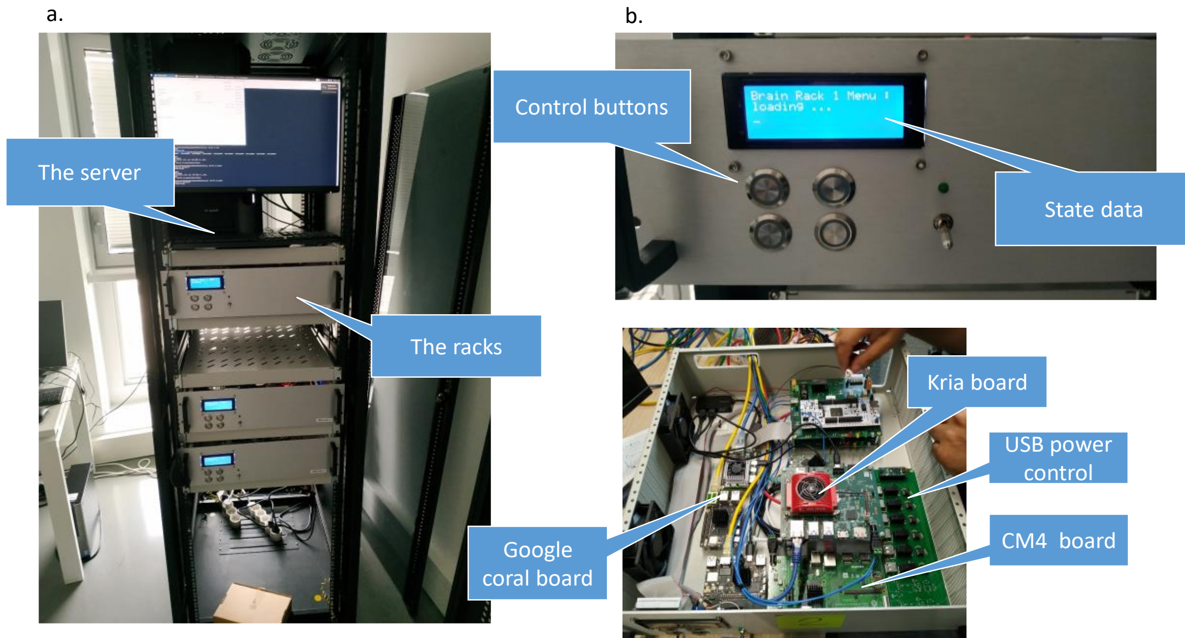

Figure 11. Views of the physical test-bench, with a close-up view of one rack

D.2 The Monitoring boards

The monitoring boards aim at (i) controlling the power supply and reset lines of the target boards, and (ii) provide current and voltage measurements. In addition, there is at least one monitoring board per rack to control (i) the rack power supply, and (ii) the buttons and LCD displays (see Figure 11.c).

A picture of a populated monitoring board is given on Figure 12. The physical architecture is depicted on Figure 13.

Figure 12. Picture of a monitoring board

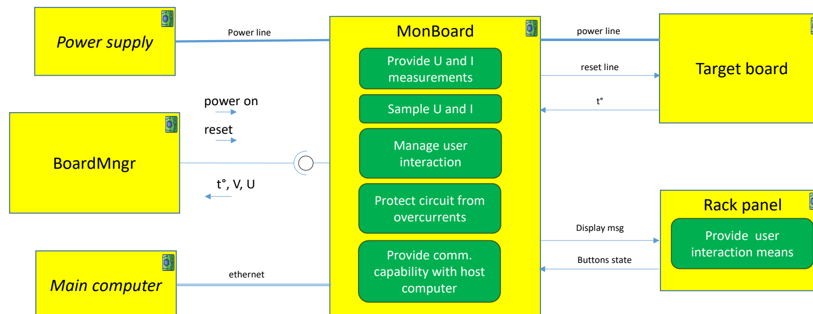

Figure 13. Logical architecture of a monitoring board

D.3 The managed USB boards

Some of the target boards are powered via USB B and C connectors. Therefore, in order to control their power and maintain the capability to communicate with these boards via USB, a dedicated board has been developed. The physical architecture of the board is shown on Figure 14.

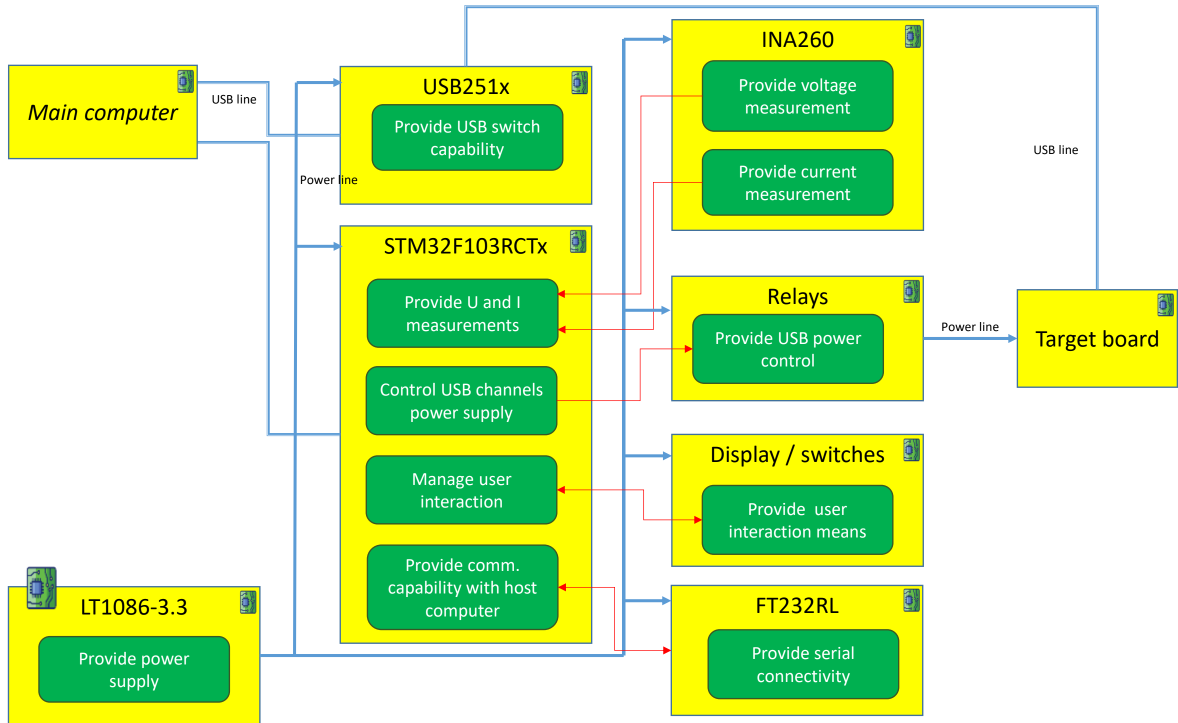

Figure 14. Physical and functional architecture of the managed USB board

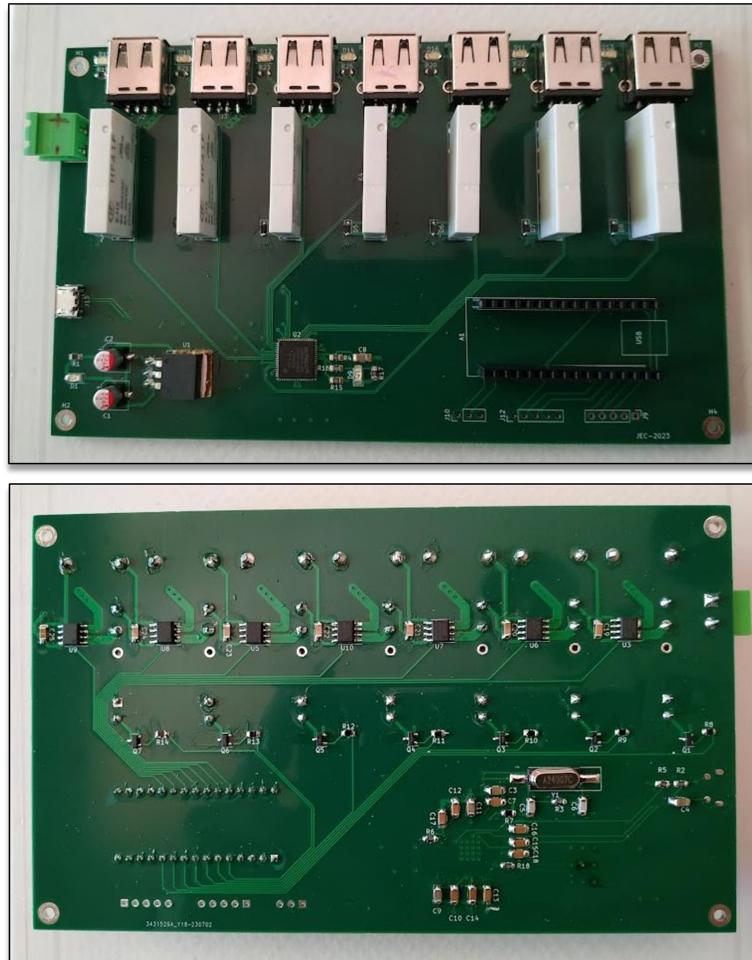

Figure 15. Managed USB board V1.0

E. Status and further developments

As of December 2023, several developments remain to be done to provide the expected level of service. In particular:

- Finalize the bench infrastructure by providing
  - full management of start/stop of containers
  - full access to configuration data
  - full management of the board electrical state (power-on / off)
  - improved error management
  - capability to access a hardware target independently of its actual physical logical
  - full reproducibility of the test bench.
- Complete the integration of the automatic generation of workflows so as to support interaction in the Jupyter notebook
  - Complete the integration of the performance prediction component for the Jetson nano and Jetson Orin targets.
    Once fully integrated, the prediction model should be substitutable usable as any (supported) real harware target. For instance, using target "jetson-orin" would deploy the model on a real Jetson Orin whereas using target "jetson-orin-pred" would use the prediction model. In addition, prediction is currently only available for the jetson nano and orin, i.e., performance can be predicted for any model on either of these platforms. CPU platforms are not yet supported. This will be done in 2024 for the arm Cortex cores (the ones used on the Orin).
    Finally, we plan to provide the capability to predict performances for various models and various hardware. This would allow the testbench to answer questions such as: “What would be the performance of that model on that hardware characterized by a certain core type and frequency, a certain memory size and frequency, etc.
- Finalize the integration of the
    - IREE toolchain
    - FINN toolchain
    - Low-level GPU toolchain (cutlass)
    - Low-level jacinto toolchain (mlib)
- Complete the Wiki documentation for the whole set of boards and tools
- Exercise the complete infrastructure on a full experimental scenario

F. Bibliography

[S. Leroux, P. Simoens, M. Lootus, K. Thakore and A. Sharma], "TinyMLOps: Operational Challenges for Widespread Edge AI Adoption," in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Lyon, France, 2022 pp. 1003-1010.
doi: 10.1109/IPDPSW55747.2022.00160

Title : Architecture of the Confiance.ai ML performance evaluation bench

Keywords : Performance evaluation bench, machine learning, optimization

Choosing a hardware target and toolchain for the implementation of AI algorithms is a highly complex task. This complexity is due to the diversity and specificity of targets and the challenges of installation, usage, and sometimes the lack of maturity of the software tools required for the implementation and deployment of these algorithms.

In this context, the Machine Learning evaluation bench developed as part of the Confiance.ia program aims to facilitate the implementation and evaluation of ML models by providing (i) a set of readily accessible boards, (ii) a set of pre-installed and configured software tools, and (iii) accompanying documentation.

This document provides an overview of the general architecture of the bench and its main components.

Titre : Architecture du banc d'évaluation de performances ML de Confiance.ia

Mots clefs : Banc d'évaluation de performance, machine learning, optimisation

Effectuer le choix d'une cible matérielle et d'une chaîne d'outils pour la mise en œuvre d'algorithme IA est une tâche très complexe. Cette complexité est due à la diversité et la spécificité des cibles et aux difficultés d'installation, d'utilisation et parfois au manque de maturité des outils logiciels nécessaires à l'implémentation et aux déploiement de ces algorithmes.
Dans ce contexte, le banc d'évaluation de Machine Learning développé dans le cadre du programme Confiance.ia a pour objectif de faciliter la mise en œuvre et l'évaluation de modèles ML en apportant (i) un ensemble de cartes directement accessibles, (ii) un ensemble d'outils logiciels préinstallés et configurés, (iii) une documentation associée.
Ce document donne un aperçu de l'architecture générale du banc et de ses principaux composants.

Our partners