Horizon 2020 Program (2014-2020)

A computing toolkit for building efficient autonomous applications leveraging humanistic intelligence (TEACHING)

D2.2: – *Refined requirement specifications and preliminary release of the computing and communication platform* †

Contractual Date of Delivery	31/08/2021
Actual Date of Delivery	28/09/2021
Deliverable Security Class	Public
Editor	*Massimo Coppola (CNR)*
Contributors	CNR: Pietro Cassarà, Massimo Coppola, Alberto Gotta, Patrizio Dazzi, Emanuele Carlini UNIPI: Gabriele Mencagli HUA: Konstantinos Tserpes TRT: Sylvain Girbal IFAG: Jakob Valtl
Quality Assurance	*Konstantinos Tserpes (HUA)*

† *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871385.*

The TEACHING Consortium

University of Pisa (UNIPI)	Coordinator	Italy
Harokopio University of Athens (HUA)	Principal Contractor	Greece
Consiglio Nazionale delle Ricerche (CNR)	Principal Contractor	Italy
Graz University of Technology (TUG)	Principal Contractor	Austria
AVL List GmbH	Principal Contractor	Austria
Marelli Europe S.p.A.	Principal Contractor	Italy
Ideas & Motion	Principal Contractor	Italy
Thales Research & Technology	Principal Contractor	France
Information Technology for Market Leadership	Principal Contractor	Greece
Infineon Technologies AG	Principal Contractor	Germany

Document Revisions & Quality Assurance

Internal Reviewers

(i) Konstantinos Tserpes, (HUA)

Revisions

Version	Date	By	Overview
1.0.0	28/09/2021	Editor	Final
0.9.0	23/09/2021	Reviewer	Comments on draft
0.2.0	22/09/2021	Editor	Completed Second draft.
0.1.2	21/09/2021	Jakob Valtl (IFAG), Sylvain Gerbail (TRT), Massimo Coppola (CNR), Pietro Cassarà (CNR), Aberto Gotta (CNR), Patrizio Dazzi (CNR), Emanuele Carlini (CNR), Konstantinos Tserpes (HUA)	Improved First draft, revised structure.
0.1.1	15/09/2021	Jakob Valtl (IFAG), Gabriele Mencagli (UNIPI),	Improved First draft.
0.1.0	08/09/2021	Gabriele Mencagli (UNIPI), Sylvain Girbal (TRT), Massimo Coppola (CNR)	First draft, revised ToC
0.0.2	30/07/2021	Contributors	Comments on the ToC.
0.0.1	08/07/2021	Editor	ToC.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AIaaS	Artificial Intelligence as a Service
CBB	Cyber Black-Box
CPS	Cyber Physical System
CPSoS	Cyber Physical Systems of Systems
D2D	Device-to-Device
DCNN	Deep convolutional neural network
DT	Digital Twin
EC	European Commission
eNB	LTE evolved Node BaseStations.
HPC2I	High-Performance Computing and Communication Infrastructure
ICN	Information-Centric Networks
IoT	Internet of Things
ITS	Intelligent Transport System
MEC	Multi-Access Edge Computing
NFV	Network Functions Virtualization
NN	Neural Network
pub/sub	publish/subscribe
QoS	Quality of Service
RAM	Range Angle Map
RDM	Range Doppler Map
RSI	Road-Side Infrastructure
RSU	Road-Side Unit
SDN	Software-Defined Networks
UW2C	User-Wide-Wearable-Collection
V2X	Vehicle-to-Everything
LTE	Long-Term Evolution
WP	Work Package
LM	Learning Module
FPGA	Field Programmable Gate Array
MQTT	MQ Telemetry Transport
LiDAR	Light Detection and Ranging

Executive Summary

This deliverable is aimed at providing an updated report on state-of-the-art analysis, conceptual design and implementation of the Distributed Computing and Communication platform for CPSoS, that we refer as High-Performance Computing and Communication Infrastructure (HPC2I).

After providing an update on the State-of-the-art for some of the related research activities we summarize the overall design and features of the HPC2I. We present a more concrete and detailed design of the conceptual, high-level system architecture previously described at M12. Said infrastructure lays the foundation of the TEACHING platform and supports the execution of project use cases.

This deliverable also focuses on the implementation of the different software tools and prototypes that compose the HPC2I, highlighting the relationships they have with the work package activities (tasks). For each of these tools is also provided the rationale and a description of the base technologies on top of which the actual software components are developed.

1 Introduction

This document is the report describing the demonstrators and prototypes developed at M18 in the context of the TEACHING project, by the WP2, *"Distributed Computing and Communication platform for CPSoS"*. This deliverable contains refinements on the requirement specifications as well as any changes to the overall design and implementation details on the preliminary release of the computing and communication platform of TEACHING. The deliverable discusses the results produced by all WP2 activities, presenting their rationale, the research results, and where appropriate an evaluation of the corresponding demonstrator.

The aim of WP2 is to achieve a platform design, develop its implementation and demonstrate it over the TEACHING use cases. The first design of the *High-Performance Computing and Communication Infrastructure* platform (HPC2I) was presented in D2.1.

As it was already reported in the M18 progress report, mainly due to Covid-related issues (collaboration impairment, recruiting impairment) at M18 the WP2 only achieved a partial integration. However, all tasks gave priority to and have achieved their own internal milestones. Some tasks have produced prototype tools which are already in use by other WPs, so they are fully on track toward project integration. Other tasks by M18 have produced demonstrators that are functional but are not yet integrated within the HPC2I platform, and the path towards integration is discussed in these cases.

The present Section 1 provides and introduction to the deliverable and summarizes the main relationships with other project deliverables.

Section 2 contains updates on the state of the art for some of the active tasks. Unlike in previous WP2 deliverable D2.1, there is no specific section for requirement updates as there were only a couple. The very few updates/fixes to previously identified requirements are in Section 4.1.1.

Section 3 relates on the overall HPC2I design, building on the initial definition provided in D2.1 and describing high level and design issues and choices. Section 4 provides instead more technical and implementation details on the demonstrator prototypes. Finally, all the subsections of Section 4 contain an evaluation of the prototype performances and features.

Section 5 draws conclusions on the deliverable content.

Section 6 of the deliverable is an Appendix, as it contains a short tutorial on the Kafka-MQTT bridge client described in Section 4.5, that is employed by the HPC2I platform and by the WP4 AIaaS platform (see Section 3 of this deliverable, and further information in deliverable D4.2

1.1 Relationship with other deliverables

D2.2 is the evolution and continuation of the D2.1 delivered at M12, which presented the state-of-the-art and requirements for the TEACHING computing platforms (HPC2I). This deliverable builds on those to provide the overall design and features of the HPC2I.

This deliverable is part of the milestone "*MS2: First integrated setup with mock-up of the TEACHING platform".* MS2 includes four more deliverables, whose relationship with D2.2 are detailed in Table 1 below.

Deliverable	Title	Relationship with D2.2

D1.2	TEACHING CPSoS architecture and specifications	D2.2 exploits the overall design and concepts of TEACHING in D1.2 to deliver implementations that are tailored to the project use cases.
D3.2	Interim Report on Engineering Methods and Architecture Patterns of Dependable CPSoS	D2.2 provides the computational framework as input for the D3.2.
D4.2	Report on integrated mockup of the AIaaS system	The AIaaS designed in the WP4 is a central part of the TEACHING platform. The interoperation between the AIaaS platform and the HPC2I is described in Section 3.2.1.
D5.2	Preliminary use case deployment, implementation and integration report with related dataset release	Several technologies that are used in the HPC2I are tailored and customized specifically for the use cases requirements.

Table 1: Deliverable grouping for verification of TEACHING Milestone 2

In general, for the MS2 milestone, the intention is to update the content of the body of knowledge of deliverables Dx.1 and present incremental improvements and mock-up functionalities of the systems. The mapping of the viewpoints of the different WPs and deliverables is represented in Figure 1.

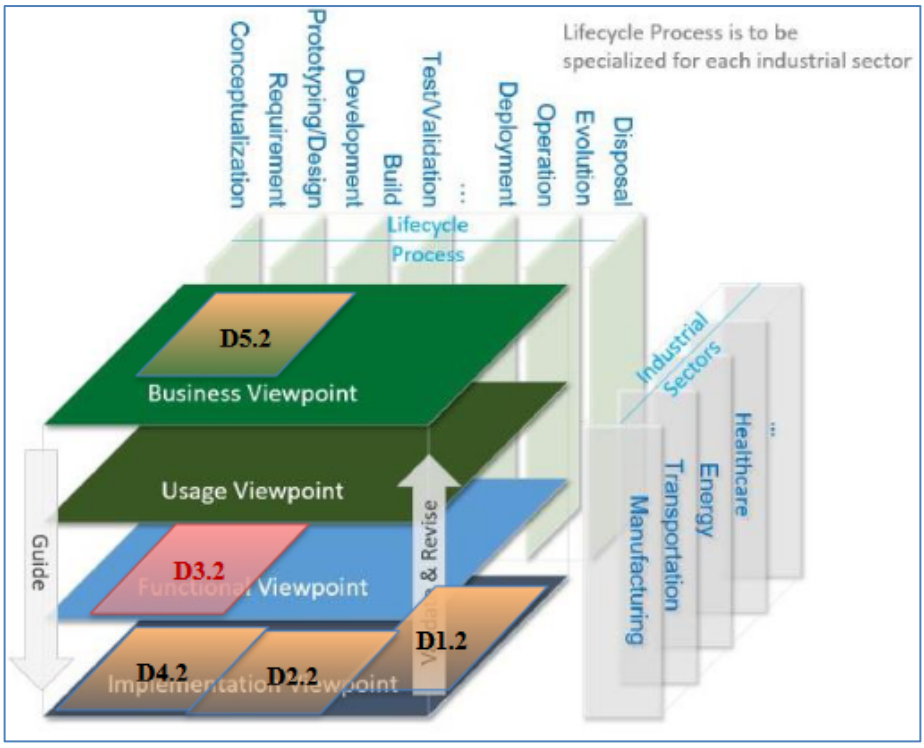


Figure 1: Depiction of the mapping of TEACHING Deliverable focuses

2 State-of-the-art Analysis Update

This section reports significant updates to the State-of-the-Art analysis reported in D2.1 for some of the active tasks of WP2.

Section 2.1 discusses recent results and research approaches related to task T2.2 (*Efficient Exploitation of Heterogeneous Computational Resources, Including Multi/Many Cores CPUs, Computing Accelerators and FPGAs*).

Section 2.2 surveys recent results related to T2.3 (*High Performance Processing and Management of Data Streams*).

Section 2.3 presents recent results related to the Vehicle-to-anything (V2X) approach which where relevant for the research work done in Task 2.5 (*Efficient and decentralized information exchange within single CPSoS and across different CPSoSs*) and is referenced by the TEACHING V2X networking that was implemented in the task and is described later on in Section 4.5 of this document.

Section 2.4 points at relevant literature about the MultiAccess Edge Computing (MEC) approach, which is relevant for the HPC2I overall design (Section 3) as well as for tasks T2.5 and T2.6 (*Seamless application deployment in Cloud and Edge resources for the distributed provisioning of computing capacity*).

Section 2.5 presents updates to the state-of-the-Art related to task T2.7 (*Silicon-born dependable AI*) referenced by the description of task T2.7 prototype in Section 4.8.

2.1 Efficient Exploitation of Heterogeneous Computational Resources

Data Stream Processing on heterogeneous resources including GPUs and FPGAs is a promising research direction to speedup streaming tasks in case of strict real-time constraints[1]. However, few research works have been investigated this research direction in depth. We mention below some research activities that have been analyzed during the refinement of the state-of-the-art analysis conducted by T2.2 *(Efficient Exploitation of Heterogeneous Computational Resources, Including Multi/Many Cores CPUs, Computing Accelerators and FPGAs*).

In stream processing applications, streaming queries ingest huge amount of data in the form of streams. Usually, the application semantics requires to compute analytics on portion of the stream having the form of windows of different kinds (e.g., tumbling, sliding, hopping). Accelerating sliding-window computations is therefore a critical task that might benefit from the offloading on a coprocessor, if available. Another work[2] presents an efficient and scalable FPGA-based acceleration of sliding-window aggregates, an important family of computations applying an associative operator over all the inputs received falling in the same window content. When windows are overlapping, a huge number of windows can trigger very frequently, raising the computational burden of such kind of operators. The FPGA implementation proposed in this paper applies a combination of previously studied techniques: *window identifiers* and

[1] Shuhao Zhang, Feng Zhang, Yingjun Wu, Bingsheng He, and Paul Johns. 2020. Hardware-Conscious Stream Processing: A Survey. SIGMOD Rec. 48, 4 (December 2019), 18–29.

[2] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie and T. Yoshinaga, "An Efficient and Scalable Implementation of Sliding-Window Aggregate Operator on FPGA," 2013 First International Symposium on Computing and Networking, 2013, pp. 112-121.

panes[3]. The latter is used to avoid replicating many aggregation modules for overlapping sliding windows, by dividing each window into non-overlapping sub-windows called panes. The implementation has been evaluated using a Xilinx FPGA and directly developed using VHDL.

Another work following the same direction has been proposed[4]. The approach is based on a Maxeler's Dataflow Engine and a deep pipeline capable of providing throughput of millions of inputs per second for the considered benchmarks featuring millions of active keys and showcasing a latency of few microseconds.

In addition to aggregates, other streaming operators have been studied to develop an FPGA implementation. An important candidate is represented by *stream joins*, a notoriously computationally expensive stateful streaming operator. An FPGA implementation has been proposed with the name of *StreamJoin*[5] targeting the Convery HC-2ex hybrid computer featuring an Intel CPU and a Xilinx FPGA. The approach has been compared against the major streaming join implementation presented in the literature (e.g., HandShake Join, ScaleJoin and CellJoin) outperforming those solutions by one order of magnitude in throughput.

A generic high-throughput architecture for stream processing targeting FPGAs has been proposed[6]. It is oriented to data stream mining problems, but it covers join operators and their porting on FPGAs. The work converged into the so-called *StreamJoin* proposal, used as a baseline for the research work already discussed.

2.2 High-Performance Processing and Management of Data Streams

We report the refinement of the state-of-the-art analysis conducted during the last months related to the T2.3 activities. Since this task is aimed at investigating efficient solutions for processing and managing data streams, we will target in this description two separated topics not previously included in the first version of the analysis. The first is related to scheduling of stream processing applications on multicores, while the second is centered around autonomic self-adaptive runtime systems for streaming.

Scheduling of Stream Processing Applications

Streaming applications are designed as data-flow graph of operators, each to be executed on the underlying computing resources (e.g., cores of multicore-based computing platforms). The standard approach adopted in traditional frameworks like Apache Storm and Flink is to run each operator by dedicated threads, each scheduled on top of the underlying commodity Operating System (e.g., Linux or Unix).

The way in which operators are scheduled might have a significant impact on the performance, and this is exacerbated on resource-constrained devices like embedded architectures, where

[3] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD Rec. 34, 1 (March 2005), 39–44.

[4] P. R. Geethakumari, V. Gulisano, B. J. Svensson, P. Trancoso and I. Sourdis, "Single window stream aggregation using reconfigurable hardware," 2017 International Conference on Field Programmable Technology (ICFPT), 2017, pp. 112-119, doi: 10.1109/FPT.2017.8280128.

[5] C. Kritikakis, G. Chrysos, A. Dollas and D. N. Pnevmatikatos, "An FPGA-based high-throughput stream join architecture," 2016 26th International Conference on Field Programmable Logic and Applications (FPL), 2016, pp. 1-4, doi: 10.1109/FPL.2016.7577354.

[6] C. Rousopoulos, E. Karandeinos, G. Chrysos, A. Dollas and D. N. Pnevmatikatos, "A generic high throughput architecture for stream processing," 2017 27th International Conference on Field Programmable Logic and Applications (FPL), 2017, pp. 1-5, doi: 10.23919/FPL.2017.8056796.

cores and CPU clock cycles must be carefully utilized. For this reason, over the years several solutions have been proposed to customize the scheduling of streaming operators of one or more co-running applications on top of the same computing platform. Haren[7] is a customizable user-space scheduler based on the Liebre streaming framework[8]. As a user-space scheduler, it is tightly coupled with the Liebre framework and cannot be adapted transparently to other more popular streaming systems. EdgeWise[9] adopts a similar solution for Storm. The Storm runtime system has been redeveloped to introduce a fixed-size pool of kernel-level threads, while operators are logical entities (user-space threads) mapped dynamically on top of the kernel-level threads by a custom user-space scheduler using a fixed policy (called Congestion-Aware Scheduling). EdgeWise cannot be added to different streaming systems (or different versions of Storm) without significant changes in its runtime environment.

To control the resource utilization of streaming operators, several low-level mechanisms provided by the Linux OS can be used and regulated. *Cgroups* have been used in Storm for coarse-grained control of resource allocation, and in resource controllers[10], to adjust the computational capacity of each application in a shared platform and reduce QoS violations. In terms of policies, i.e., deciding how frequently operators should be put in execution and for how long, prior works have investigated different solutions aimed at privileging throughput or latency. Examples are the Queue-Size policy (QS), Highest-Rate (HR)[11], and FCFS[12]. The Chain policy[13] minimizes the memory usage of multiple applications and can take maximum query latency into account. Since those policies are particularly effective to run efficiently streaming applications on resource-constrained devices like the ones of the TEACHING infrastructures, they represent important solutions to be studied and adapted to the project's needs.

Self-adaptive Streaming Runtimes

To keep with changing workloads, streaming frameworks should adapt the runtime system behavior and resource utilization to fit the *Quality of Service* requirements expected by clients/users.

Several entities and configuration parameters can be adapted at runtime[14]. *Batching* can be used as an optimization in some application scenarios to enhance throughput at the expense of

[7] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatri-antafilou. 2019. Haren: A Framework for Ad-Hoc Thread SchedulingPolicies for Data Streaming Applications. InProceedings of the 13thACM International Conference on Distributed and Event-Based Systems(DEBS '19). ACM, Darmstadt, Germany, 19–30.

[8] Open Source. 2021. Liebre SPE. Retrieved March 11, 2021 fromhttps://github.com/vincenzo-gulisano/Liebre.

[9] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. 2019.Edgewise: A Better Stream Processing Engine for the Edge. InUSENIXAnnual Technical Conference (ATC) 19. USENIX, WA, USA, 929–946.

[10] M. Reza HoseinyFarahabady, Ali Jannesari, Javid Taheri, Wei Bao, Al-bert Y. Zomaya, and Zahir Tari. 2020. Q-Flink: A QoS-Aware Controllerfor Apache Flink. In2020 20th IEEE/ACM International Symposium onCluster, Cloud and Internet Computing (CCGRID). IEEE, Melbourne, VIC,Australia, 629–638.

[11] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. 2019.Edgewise: A Better Stream Processing Engine for the Edge. InUSENIXAnnual Technical Conference (ATC) 19. USENIX, WA, USA, 929–946.

[12] Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, andKirk Pruhs. 2008. Algorithms and Metrics for Processing MultipleHeterogeneous Continuous Queries.ACM Transactions on DatabaseSystems33, 1 (March 2008), 1–44.

[13] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. 1998.Flow and Stretch Metrics for Scheduling Continuous Job Streams. InProceedings of the Ninth Annual ACM-SIAM Symposium on DiscreteAlgorithms (SODA '98). Society for Industrial and Applied Mathematics,Philadelphia, PA, USA, 270–279.

[14] H. Herodotou, Y. Chen, and J. Lu. A survey on automatic parameter tuning for bigdata processing systems.ACM Comput Surv, 53(2), 2020.

latency[15]. The number of cores and their frequency can be changed at runtime for reducing energy consumption[16], as well as the dynamic tuning of the concurrency modes adopted by concurrency queues used for data forwarding between stages[17]. There are also works dynamically changing the degree of parallelism of parallel stages[18]. However, these optimizations are not flexible or sufficient for the kinds of adaptations required by real-world stream processing applications. For example, very dynamic scenarios may require complex changes of the application structure[19] (e.g., the runtime can switch from a stream-based pattern implementation to a completely different one, having different load balancing, memory and resource consumption properties). Although interesting, such kind of complex reconfigurations are affected by semantics issues and may be detrimental to the application performance due to temporarily downtimes.

Concurrent recompilation has been proposed for reducing application downtimes[20]. However, the techniques needed for controlling downtimes are intrusive, and can affect the computational semantics (results ordering). In practice, this approach is hard to generalize. A study proposes *Grizzly*[21], a solution that encompasses adaptive compilation to change the execution at runtime. However, the code generation approach is bound to structured streaming applications based on linear algebra operators, it cannot be straightforwardly generalized to general-purpose streaming workloads needing custom user functions within the operator business logic code.

2.3 V2X Access Architecture

Vehicle-to-everything (V2X) communications represent an important niche in both networking and vehicular technologies fields.

Sattiraju et al.[22] compare link level performance of ITS-G5 standard, based on IEEE 801.11p, and the more recent Cellular-V2X (C-V2X), which is based on 3GPP standards. As stated by the authors, IEEE 802.11p standard is a well matured technology that has been researched for over 20 years. Notwithstanding this, the standard never had a deep impact in the automotive mass market, probably due to the major investment in communication infrastructure to support Intelligent Transport systems (ITSs).

[15] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive Stream Processing using DynamicBatch Sizing. InACM Symp. on Cloud Computing, pages 1–13. ACM, 2014.

[16] D. De Sensi, T. De Matteis, and M. Danelutto. Simplifying Self-Adaptive and Power-Aware Computing with Nornir.Future Gener Comput Syst, 87:136–151, 2018.

[17] M. Torquati, D. De Sensi, G. Mencagli, M. Aldinucci, and M. Danelutto. Power-awarePipelining with Automatic Concurrency Control.Concurr Comput, 31(5):e4652, 2019.

[18] V. Kalavri, J. Liagouris, M. Hoffmann, and et al. Three steps is all you need: fast,accurate, automatic scaling decisions for distributed streaming dataflows. InUSENIXOper. Systems Design and Implemen., pages 783–798, 2018.

[19] S. Rajadurai, J. Bosboom, W.-F. Wong, and S. Amarasinghe. Gloss: Seamless livereconfiguration and reoptimization of stream programs.ACM Not, 53(2):98–112, 2018.

[20] X. Liu, A. V. Dastjerdi, R. N. Calheiros, C. Qu, and R. Buyya. A Stepwise Auto-Profiling Method for Performance Optimization of Streaming Applications.ACM TransAuton Adap, 12(4):1–33, 2017.

[21] P. Grulich, B. Sebastian, S. Zeuch, and et al. Grizzly: Efficient stream processing throughadaptive query compilation. InACM SIGMOD Intl. Conf. on Management of Data,pages 2487

[22] Sattiraju, R., Wang, D., Weinand, A., & Schotten, H. D. (2020). Link level performance comparison of c-v2x and its-g5 for vehicular channel models. In *2020 ieee 91st vehicular technology conference (vtc2020-spring)* (pp. 1–7).

A comparison with legacy LTE networks was already presented in 2014[23], while the Release.12 LTE Device-to-Device (D2D) link level put the comparison of the two standards on a fairer footing. C-V2X, introduced in Release.14, sheds light on V2X communications, with the first specification in 3GPP standards and Sattiraju et al. achieve that C-V2X outperforms IEEE 802.11p for almost all the considered fading channels with a gain ranging from 0-5 dB, and, above all, at high vehicle speeds. C-V2X extends LTE's D2D communication modes by defining two operation modes for vehicular systems

1. Mode 3 within the coverage and
2. Mode 4 out of the coverage of an LTE eNB (evolved NodeBaseStations).

Mode 3 uses the LTE-Uu radio interface, whereas Mode 4 uses the new PC5 one. The maximum allowed latency ranges between 20 and 100ms, depending on the application[24].

In the HPC2I architecture we foster the application of C-V2X technology, as it will be discussed in Section 4.5, whereas C-V2X allows for faster and easier deployment of supporting communication infrastructures by simply upgrading the existing base stations for legacy services and start offering pervasive coverage on a global scale.

2.4 Multiaccess Edge Architecture

Multi-Access Edge Computing (MEC) is an emerging network architecture concept aimed at enabling cloud computing capabilities at the edge of the network. The resulting environment is characterized by ultra-low latency and high bandwidth as well as real-time access to the radio network, which can be leveraged by applications. A platform model providing access to these features perfectly matches with the TEACHING project needs, especially but not only in the case of automotive applications. We will describe in Section 4.5 how the MEC approach and its Edge-Cloud Continuum model are exploited in the HPC2I architecture. Here we will just mention that allowing some computational tasks to migrate from the Edge to the Cloud and back allows several optimizations (e.g., in terms of local overhead and service latency) that can improve both the HPC2I platform efficiency and the end-user QoE.

Some authors[25],[26] provide an overview of existing MEC-based infrastructures, analyzing communication and computational issues. The core of MEC infrastructure is based on a virtualized platform that leverages recent advancements in Network Functions Virtualization (NFV), Information-Centric Networks (ICN), and Software-Defined Networks (SDNs). NFV allows a single edge node to provide multiple and differentiated computing services, by instantiating multiple virtual machines for performing different operations. ICN provides an alternative end-to-end service recognition paradigm for MEC, shifting from a network-centric to an information-centric paradigm for implementing context-aware computing. Lastly, SDN allows MEC infrastructure administrators to manage services via function abstraction, achieving scalable and dynamic computing.

---

[23] Moller, A., Nuckelt, J., Rose, D. M., & Kurner, T. (2014). Physical layer performance compar- ison of lte and ieee 802.11 p for vehicular communication in an urban nlos scenario. In *2014 ieee 80th vehicular technology conference (vtc2014-fall)* (pp. 1–5).

[24] Marojevic, V. (2018). C-v2x security requirements and procedures: Survey and research directions. *arXiv preprint arXiv:1807.09338*.

[25] Mao, Y., You, C., Zhang, J., Huang, K., & Letaief, K. B. (2017, November). A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys Tutorials*, *19*(4), 2322-2358.

[26] Mach, P., & Becvar, Z. (2017, March). Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communication Surveys Tutorials*, *19*(3), 1628-1656.

Many pieces of evidence are demonstrating the contribution of the technologies to certain classes of application to improve the Quality of Experience (QoE) for end-users. Some authors[27],[28] propose a resource allocation method for a mobile multi-end user scenario, where computation tasks can be split simultaneously in local computing and offloading. The local computing is performed at the edge nodes and the offloading toward an aggregator in the Cloud. At the Edge, information is collected, e.g., on energy consumption, channel gains, and fairness factors. This information is aggregated in the Cloud to estimate the global optimal strategy for the allocation resources. The authors provide an interesting analysis of the performance obtained by the proposed methods where parameters such as latency, computation time, and limited hardware resources are not negligible parameters.

Other authors[29],[30],[31] propose examples of MEC-based infrastructures for vehicular scenarios. Different solutions are explored for the scheduling of service operations started by vehicles, using either offloading approaches or cooperative ones. In the first case, the proposed infrastructure allocates the resources to address the operations at the edge nodes, if a sufficient capacity is available; otherwise moves the service operations to the Cloud. In the second case, MEC is used as an enabling communication platform that support direct interactions between vehicles, opening chances to cooperative collaboration schemes.

2.5 Resilient, Fail-Safe and Energy-Efficient, Silicon Born AI

Autonomous mobility and its components are an emerging research field. Numerous researchers are working currently on topics related to Artificial Intelligence (AI) as well as system strategies to enable an affordable, safe, and secure fully autonomous diving (level 5). Different sensor types offer various sensor-specific benefits as well as limitations.

Especially radar sensors are of great interest for autonomous driving thanks to their capability to operate even under bad weather conditions. Radar sensors do not have mechanically moving parts, unlike other sensors that give a depth information e.g., LiDAR, thus being physically more robust. Finally, the price advantage of a radar system in comparison to a LiDAR system might give radar sensors an advantage in future autonomous vehicles.
Nevertheless, there are still some challenges that need to be resolved, like the complicated and time-consuming data processing. The aforementioned challenges are addressed by using a fast neural network-based approach.
The evaluation of incoming sensor data must happen in a predefined period to ensure a response of an autonomous vehicle due to a changing environment. If the time constrains cannot be met, the safety of the vehicle as well as surrounding pedestrians or other traffic participants cannot be ensured. AI based evaluation methods are one possibility to deal with the vast amount of data in the required time constrains.

---

[27] Wang, C., Liang, C., Yu, F. R., Chen, Q., & Tang, L. (2017, August). Computation offloading and resource allocation in mobile edge computing. *IEEE Trans. on Wireless Communica- tions*, *16*(8), 4924-4938.

[28] You, C., Huang, K., Chae, H., & Kim, B.-H. (2018, April). Energy-efficient resource allocation for mobile-edge computation offloading. *IEEE Trans. on Wireless Communications*, *66*(4), 1594-1608.

[29] Xie, R., Tang, Q., Wang, Q., Liu, X., Yu, F. R., & Hu, T. (2019, December). Collaborative Vehicular Edge Computing Networks: Architecture Design and Research Challenges. *IEEE Access*, *7*, 178942-178952.

[30] Wang, J., Feng, D., Zhang, S., Tang, J., & Quek, T. Q. (2019, May). Computation offloading for mobile Edge computing enabled vehicular networks. *IEEE Access*, *7*, 62624-62632.

[31] Luo, G., Yuan, Q., Zhou, H., Cheng, N., Liu, Z., Yang, F., & Shen, X. S. (2018, April). Cooperative vehicular content distribution in edge computing assisted 5G-VANET. *China Communication*, *15*(7), 1-17.

Mobile data bandwidth limits and latency issues that clash with the aforementioned time constrains only admit data processing to run on the edge devices, but do not allow a centralized data processing solution. To keep processing demand on the edge devices low, AI algorithms can be advantageous compared to traditional data processing because they require less hardware resources as they can easily be run on specialized hardware. Even an in-silicon implementation is possible as planed in this project.
Challenges that arise from the use of AI components are their black box behavior and their vulnerability towards adversarial attacks. Both can lead to unforeseen reactions of the vehicle. While many investigations can be and have been performed using simulation tools, actual testing under real world conditions is also crucial for development.
Many research efforts have been reported in the visual domain regarding test environments in the form of reduced size vehicles. Though other test environments are vision based, they focus on a variety of aspects like safety, security, or efficiency. An example of reinforcement learning was implemented by Qi z. et al.[32] on a DonkeyCar[33] platform. T. Do et al.[34] used Deep Convolutional Neural Networks (DCNN) on a monocular vision-based prototype. Both report high accuracy rates on the steering and speed decision. J. Newman et al.[35] introduced a low budged self-driving model car with the focus of maneuvering through an indoor race track.
Traditional platooning approaches [36], [37] that are like the "follower" use case rely on the communication between vehicles and use radar sensor input only for the distance detection, while we also extract the angular information of the preceding vehicle, needed for the steering from the radar data.
Data processing of radar data is a major topic in the research community with numerous publications for different purposes, reaching from gesture recognition[38] and air writing[39] to people counting[40] and human activity classification[41]. Even small-scale movements like vital signs[42] can be detected by this technology. All these applications have in common that they are performed indoor in a shielded environment and with a stationary radar sensor. Installation in a vehicle adds additional challenges due to an outdoor environment with a non-stationary

[32] Zhang Qi and Du Tao. “Self-driving scale car trained by Deep reinforcement Learning.” In: ArXiv abs/1909.03467 (2019)

[33] The DonkeyCar Website. (Accessed: 10.09.2021). URL: https://www.donkeycar.com/

[34] Q. Dang T. Do M. Duong and M. Le. “Real-Time Self-Driving Car Navigation Using Deep Neural Network”. In: 2018 4th International Conference on Green Technology and Sustainable Development (GTSD), Ho Chi Minh City, 2018 abs/1909.03467 (2018), pp. 7–12. DOI: 10.1109/GTSD.2018.8595590.

[35] J. Newman, Z. Sun, and D. -J. Lee. “Self-Driving Cars: A Platform for Learning and Research”. In: 2020 Intermountain Engineering, Technology and Computing (IETC). 2020, pp. 1–5. DOI: 10 . 1109 / IETC47856.2020.9249142.

[36] Sadayuki Tsugawa, Shin Kato, and Keiji Aoki. “An automated truck platoon for energy saving”. In: 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems. 2011, pp. 4109–4114. DOI: 10.1109/IROS.2011.6094549.

[37] Eric Chan. “SARTRE Automated Platooning Vehicles”. In: June 2016, pp. 137–150. ISBN: 9781786300270. DOI: 10 . 1002 / 9781119307785.ch10.

[38] Mateusz Chmurski and Mariusz Zubert. “Novel Radar-based Gesture Recognition System using Optimized CNN-LSTM Deep Neural Network for Lowpower Microcomputer Platform”. In: (2021).

[39] M. Arsalan, A. Santra, and V. Issakov. “Radar Trajectory-based Air-Writing Recognition using Temporal Convolutional Network”. In: 2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA). 2020, pp. 1454–1459. DOI: 10.1109/ICMLA51294.2020.00225.

[40] Cem Yusuf Aydogdu et al. “Multi-modal cross learning for improved people counting using short-range FMCW radar”. In: 2020 IEEE International Radar Conference (RADAR). IEEE. 2020, pp. 250–255.

[41] Michael Stephan et al. “Radar Image Reconstruction from Raw ADC Data using Parametric Variational Autoencoder with Domain Adaptation”. In: 25th International Conference on Pattern Recognition. IEEE. 2021.

[42] M. Arsalan, A. Santra, and C. Will. “Improved Contactless Heartbeat Estimation in FMCW Radar via Kalman Filter Tracking”. In: IEEE Sensors Letters 4.5 (2020), pp. 1–4. DOI: 10.1109/LSENS.2020. 2983706.

sensor. Besides, the tough real-time constrains implicate further challenges to the speed of evaluation.
In the "follower" use case an autonomous miniaturized vehicle follows a preceding vehicle by means of an algorithm that evaluates the information coming from the radar sensor. Details of the implementation are given in Section 5.7.

3 HPC2I Design

This section documents the overall design of the HPC2I architecture devised within WP2 and supporting the overall TEACHING needs. No significant design changes were made to the initial design documented in deliverable D2.1, but some implementation details have been refined and integrated in the design because of the development activity. While in this section we focus on the high-level design and on abstract model of the HPC2I structure, the following Section 4 provides information about the actual implementation activities.

Section 3.1 surveys the overall HPC2I architecture already described in D2.1, section 4.

Section 3.2 focuses on application support within HPC2I. The section specifically deals with (1) the interoperation with the AIaaS subsystem developed by WP4 and (2) the support HPC2I can provide to distributed applications that need to exploit federated / distributed learning.

Section 3.3 provides updates to the description of the information flow in one of the TEACHING use cases from D2.1 Section 4.

3.1 General HPC2I Platform Architecture

The High-Performance Computing and Communication Infrastructure (HPC2I) has been designed and presented in previous deliverable D2.1, with the aim of supporting the deployment and execution of autonomous, adaptive, and dependable CPSoS applications as they are developed by the TEACHING project approach. The high-level composition of Cloud-Edge Continuum platforms is depicted in Figure 2. While in the generic approach to Continuum computing shown in the Figure it is often enough to distinguish among Cloud-based resources

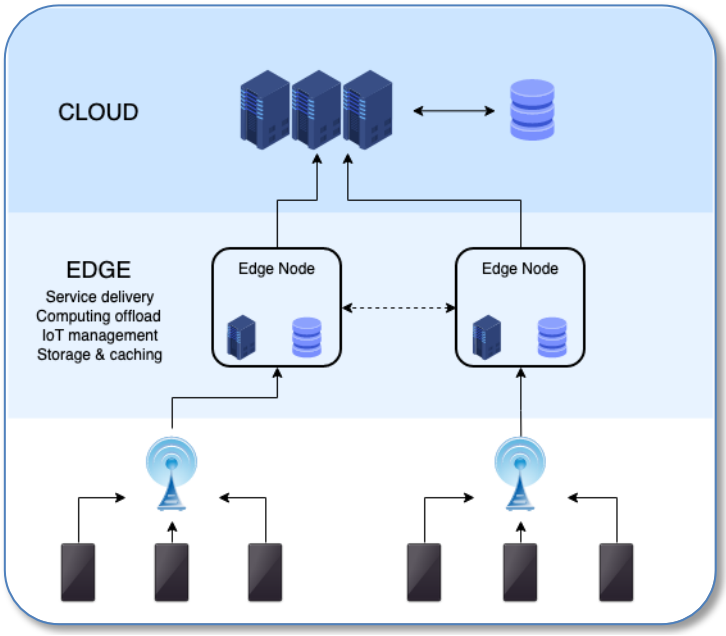


Figure 2: High-level Organization of Resources in a Cloud Edge Continuum

and services, Edge-based ones, and end-user, typically mobile devices, the aim of supporting CPSoS which leverage AI over possibly sensitive data requires use a more in-depth analysis.

We defined the HPC2I platform architecture (shown in Figure 3, taken from D2.1 with minor revisions applied) by analyzing the three main platform layers (*Cloud*, *Near Edge,* and *Far Edge*) according to different aspects of vertical composition, which are represented in the Figure in left-to-right order.

- The entities and services interacting
  - Cloud-base services, Near-Edge services and concentrators, end-user applications (including clients for centralized and distributed applications as well as local components of distributed applications)
- The type of data that those entities possess and may share
  - End-user devices on the Far edge will need to distinguish between private and shareable data (resulting from aggregation, anonymization, abstraction techniques and with user consent), while the upper levels of the service architecture will typically be tasked with federating the shareable data items.
- The type of AI models that each layer can develop
  - The TEACHING approach needs to support exploiting both local AI solutions and global one, where distributed and federated techniques will allow to extract further insight from the global collection of shareable data
- The type of physical devices and ownership of resources that apply to each layer.
  - It is a key element of TEACHING that the AI exploitation can dynamically exploit all physical device layers. This includes (public and private) Cloud resources and several possible types of Near Edge devices, provided either by telco companies (rented as part of their network infrastructure), by infrastructure owners (e.g., company managing a road, building, airport or other physical infrastructure) or by public cloud providers (as geographically dispersed, smaller cloudlets). Finally, as already stated, the end user devices are also quite heterogeneous.

The organization of the platform we summarized here for the sake of readability is not significantly changed since D2.1. We refer to that deliverable for more details on the architecture rationale. Her we underline a couple of key aspects of the approach and its development within the project.

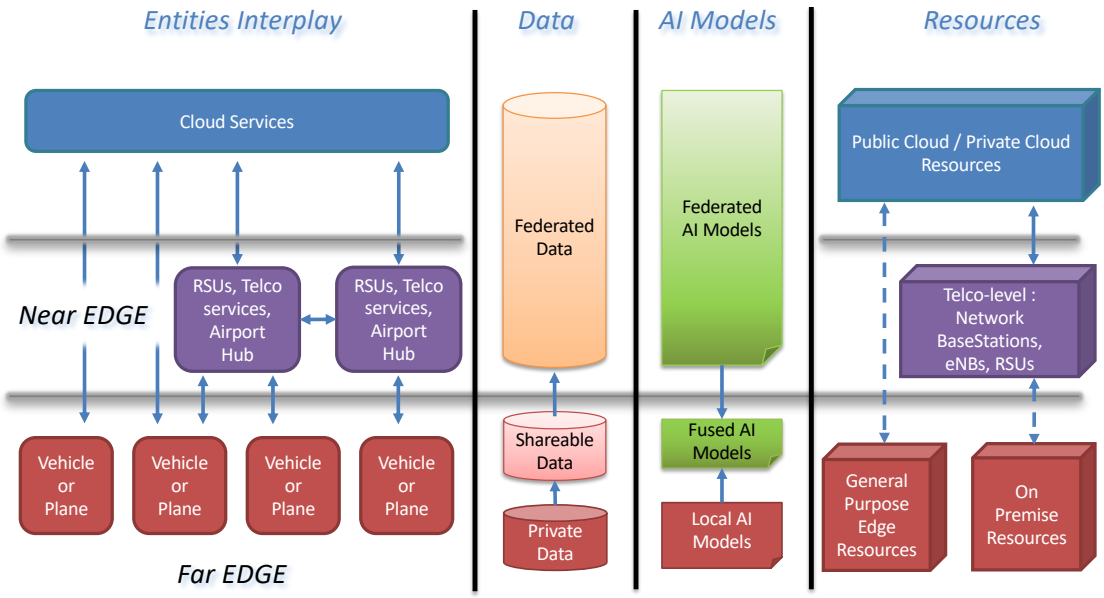


Figure 3: Conceptual Architecture of HPC2I, from D2.1

- The TEACHING approach goes along with the MEC one (described in Section 2.4), as they both deal with a Continuum of resources from multiple stakeholders and implementing multiple concurrent services. The mutual relationship among MEC and AI has been explored in the literature already, and many tools and results from the MEC approach can be leveraged by TEACHING for the HPC2I platform. We are still investigating, and we plan to continue, about the impact of the most specific constraints of our project. The key issue is to understand in what cases we can consider the HCP2I equivalent to MEC, and in what cases it is more appropriate to see HPC2I as a subset of MEC, e.g., a platform with more stringent constraints. One area of research is about the presence of real time and reliability requirements, or even just the potential interaction with part of a CPS that is subject to these requirements is one item. Another key area is the consequences in TEACHING of different distributed and federated AI models over sensitive data being carried on over the (possibly MEC compliant) HPC2I distributed platform.
- Some activities of WP2 focus on the specific needs of the Far Edge platform layer, where heterogeneous embedded devices must be exploited with high efficiency and dynamicity while preserving the reliability of existing CPS software (particularly concerning vehicle control and safety) as well as boosting the portability of AI applications.

3.2 Application Support

The overall HCP2I architecture leverages technologies developed within the area of Continuum computing. All Edge and Cloud resources are seen as dynamically exploitable resources of a common platform where they sport each one different trade-off among features like computing (network, storage) capacity, end-user vicinity (in terms of latency and tenancy) and control over privacy-related features. Current technology sees this kind of elastic applications as composed by a set of cooperating virtual machine instances an OS container instances, described in a standard formalized language that is designed to automate most of the application lifecycle.

As we describe more in detail in Section 4.6 from the technological viewpoint, we will consider the AIaaS system developed in WP4 (see deliverable D4.2) as the smallest unit of deployment available to HPC2I. AIaaS instances, besides internally exploiting a modular approach to support AI applications, are also engineered to be deployed on specific resources as a small set of OS containers (possibly just one) and can as well be deployed within VMs on any layer of the resource Continuum. The essential tasks of resource monitoring, indexing, and selection, as well as application deployment, lifecycle management, and autonomic management are achieved to a great extent by leveraging state-of-the-art Cloud technologies. Significant standard solutions do already exist to represent and manage the overall structure of application, but the key needs of TEACHING are not the same as typical application patterns like client-server or content distribution networks.

While any coordination pattern among the distributed application parts executing on distinct AIaaS instances can be programmed within the HPC2I approach, TEACHING is obviously focusing on the coordination patterns that support the various forms of distributed learning, and specifically federated learning, that are the most relevant to AI application for CPSoS.

3.2.1 Interoperation with the AIaaS Subsystem

The TEACHING HPC2I platform will support AI apps over CPSoS by leveraging the *AI as a Service* (AIaaS) approach developed in WP4.

The most succinct way of summarizing the AIaaS approach is probably to say that it is a modular approach at building AI applications that are local to a single CPS, allowing said applications to be specified in a portable way as well as

- increasing the reuse of AI code, especially tested and verified code, within distinct applications,
- automatically exploiting known optimizations and accelerator devices on a range of different hardware resources.

The AIaaS approach relies on a support software architecture that is locally executed on Edge and Cloud devices. For a full description we refer the reader to deliverable D4.2. For the sake of the relationship between HPC2I and AIaaS a few key traits of the AIaaS implementation are particularly relevant.

- **AIaaS is designed to be portable** over a range of heterogeneous devices, and it also aims at supporting different AI frameworks (with TensorFlow and TensorFlow lite being the first two targets) in a way that is effortless for the application developer. This feature provides the HPC2I platform the option to exploit the same application code on all resource layers.
- **AIaaS can be deployed as a (set of) container (s),** as well as, obviously, within a VM. Together with previous feature, this mean that standard Cloud technology can be easily adapted to manage AIaaS based applications.
- **The TEACHING approach to networking** already within the AIaaS is based on the publish/subscribe communication model. This choice provides a tool to build higher-level, distributed application structures easily and reliably from AIaaS instances, each one typically acting on a different device and playing a different role within the distributed application.

We shall note that the design choices in the tree of topics for the pub/sub communications are quite relevant for the software engineering properties of applications based on the HPC2I and AIaaS. As the project proceeds, the topic is going to be further investigated. Currently the

approach focuses on the main aim of TEACHING, AI for CPSos, thus on the various patterns of distributed learning.

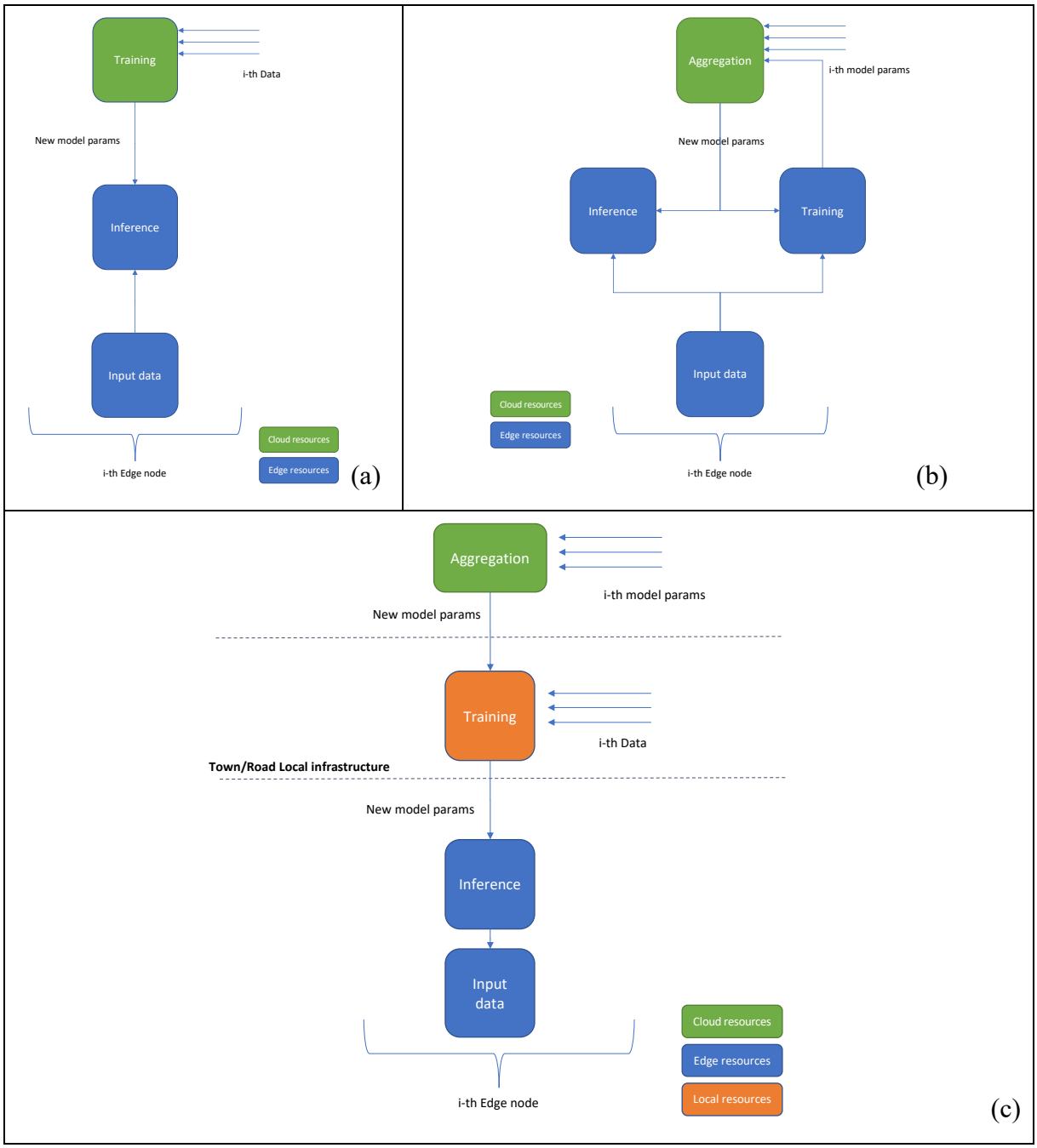


Figure 4: Centralized Learning (a), Federated learning/ Training at the edge (b), Hybrid (c)

3.2.2 Distributed and Federated Learning Support

There are scenarios in which it is beneficial or even mandatory to isolate different subsets of the training data from each other. The furthest extent of this is when a model needs to be trained on datasets that each live on different machines or clusters and may under no circumstance be

co-located or even moved. In TEACHING, we have often come across such scenario, where multiple parties jointly learn an accurate deep neural network while keeping the data itself local and confidential. We resort to a federated learning implementation to safeguard the privacy of the nodes, and the HPC2I platform supports the approach.

We distinguish 3 types of distributed learning models to supported in the TEACHING Platform: the centralized one (**Error! Reference source not found.**a), the model where training happens at the edge nodes (**Error! Reference source not found.**b) and a hybrid model where edge is distinguished based on the "distance" from the cloud core (**Error! Reference source not found.**c), i.e., the algorithm intentionally exploits both Far Edge devices and Near Edge ones.

All three patterns seen in the figure can be implemented via the HPC2I networking support, by defining appropriate topic trees that confine application communications and allow selectively retrieving data from the specific edge nodes that generate it. The pub-sub support allows both

1. simple programmed interaction among AIaaS instances – It relies on the networking and requires ad-hoc topic tree design for non-trivial patterns, where the topic structure may reflect the physical deployment structure of the application. The approach does not strongly rely on Kafka features. Scalability may be an issue for centralized or other unbalanced communication patterns, due to the tree topic pattern being encoded within the AI app.
2. Federated learning patterns – they are easily achieved by designing a topic tree according to the abstract pattern of exchange of knowledge models. As suggested by Figure, the topic trees for shareable data are separate from those of locally trained/federated models. As reported in this deliverable (Section 4.5) as well as in D4.2, the exchange of information across the boundary of and AIaaS instance (i.e., between a CPS and the whole CPSoS) is constrained by the configuration of the V2X Network bridge, that only allow specified topic trees to cross the boundary. This choice provides the option to enforce isolation among data and models, as well as among topics of different applications.

We shall mention that it is being evaluated the option to add interfaces to the AIaaS to directly exploit more advanced feature and extensions of Kafka to provide additional support to distributed learning. The required architectural change may result too complex but is being evaluated as it may also result in increased flexibility and scalability of the HPC2I platform.

Conversely, for small scale platforms it is also possible to directly exploit MQTT broker sharing. The choice to entirely skip the global networking support can reduce the deployment complexity and produce a more lightweight implementation but is currently not considered viable beyond the debugging phase, as it is not scalable.

3.3 Use Cases Information Flow

In this section we provide an update for the information flow described in D2.1 for the avionics project use case. Further information about the use cases is within D5.2.

3.3.1 Avionics Use Case Information Flow

The avionics use-case, already presented in deliverables D2.1 and D5.1, is composed of a dependable hard real-time application, the flight management system (FMS), performing the autonomous piloting, and a set of additional software components that aim at monitoring the FMS to ensure proper runtime behavior, the latter ones are collectively addressed as the *Cyber Black Box* (CBB).

The information flow external to the system embedded into the aircraft has already been presented in Deliverable D2.1 (Section 4.1.2). Beyond this external information flow, the avionics use-case also needs to deal with an internal communication flow on the embedded system itself, a flow that depends on the placement of the various components presented in Figure 5: Local information flow of the avionic use-case, on the target embedded device.

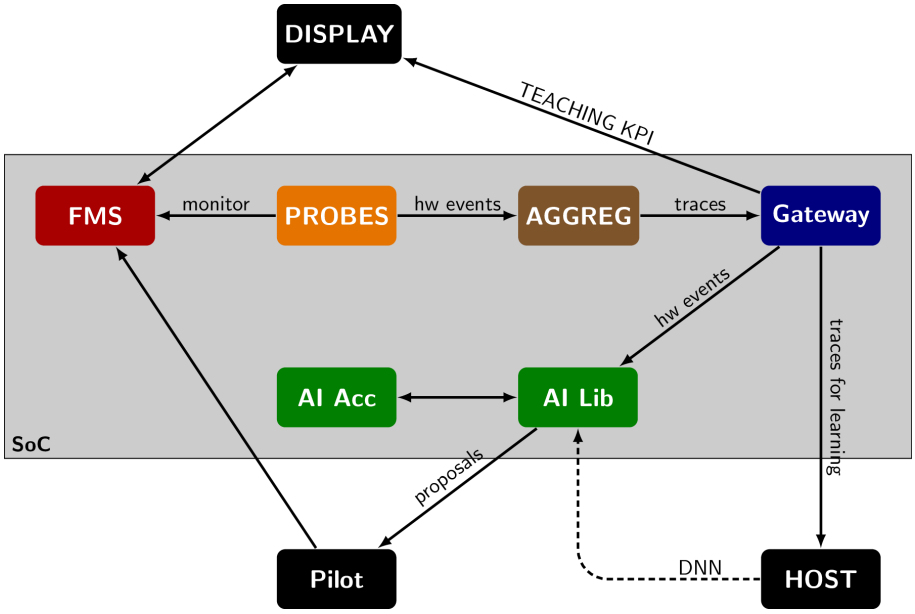


Figure 5: Local information flow of the avionic use-case

The **probes** are in charge of collecting hardware behavioral information about the **FMS** application running on the embedded system. This information has then to be **aggregated** into hardware event traces that are forwarded through a **gateway**, either to an external host possibly in the cloud to perform the machine learning of the application behavior, or to the on-board **AI accelerators** to perform the inference and detect potential threats as behavioral deviations.

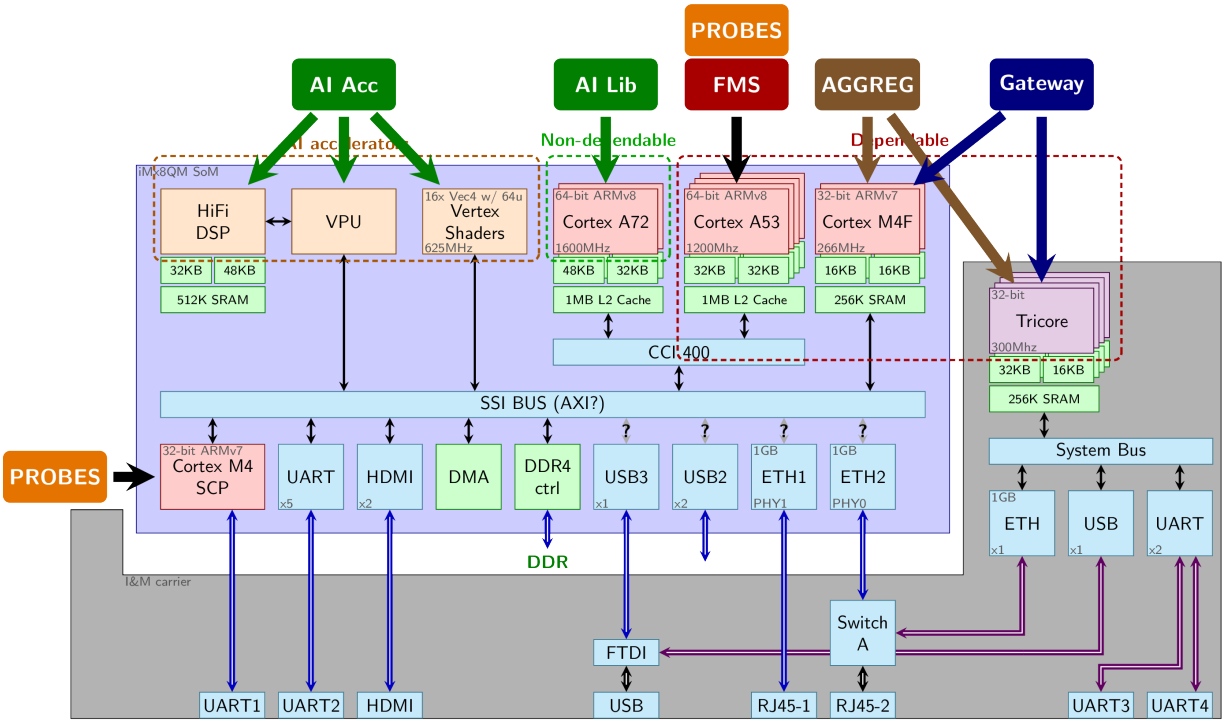


Figure 6 - Hardware mapping of the CBB components

Figure 6 illustrates a possible mapping of these software components on the target iMx8 hardware, provided by I&M. Such components are distributed among several different cores on the hardware architecture, the FMS application being mapped on the dependable Cortex A53 cores, while the inference being performed on the non-dependable Cortex A72 cores. These components are therefore using different memory hierarchies (like the private level two caches of the Cortex A72 and the Cortex A53 core clusters) but are eventually sharing the same DRAM memory and common interconnects. To guarantee the hard real-time behavior of the FMS application communication, we will have to ensure at the level of the CBB that inter-component communications will not cause extra latencies to the monitored FMS application.

Such a constraint will be ensured mostly at scheduling level by restricting some communications to specific time windows, by carefully quantifying the communication traffic, especially at the gateway and AI accelerator level.

4 Prototypes Implementation and Analysis.

This section presents the actual prototypes and tools so far developed by active WP2 tasks to support the HPC2I architecture. We shall underline that some tasks (T2.2, T2.3) produce publicly available tools, other ones (T2.4, T2.5) actually produced tools already in use within the project. Other tasks (T2.6) are developing HPC2I architecture components which have not yet been integrated, and other activities (T2.7) are still in their research phase. As a consequence, the actual features of the demonstrator associated to each activity varies, and in some cases, it is also relevant to show the experimental results (tests, or simulations) performed to guide the development and validate the prototypes.

The contents of this section are organized as follows.

Section 4.1 discusses updates to the technological choices underlying WP2 development, with respect to what was stated in project deliverable D2.1. It also details a couple of requirements that have been amended since the initial requirement analysis performed in WP2 and reported in D2.1, Section 3.

As integration of the different tools among themselves and with other project results has now a high priority, the status and plans for interoperability and integration are detailed from a technical viewpoint in section 4.2.

Sections 4.3 and 4.4 discuss the tool developed and the experiments from tasks T2.2 and T2.3. It's worth noting that for the sake of readability and to avoid duplicate definitions they are presented in reverse order in this deliverable. That is, the WindFlow programming framework developed within T2.3 is discussed first, in Section 4.3. The results discussed in Section 4.4, that focus on heterogenous hardware exploitation but also employ WindFlow, are related to Task T2.2.

Section 4.5 discusses the architecture and implementation results of the networking part of the HPC2I architecture, based on the publish/subscribe approach and on the V2X and MEC approaches discussed in Section 2.3 and Section 2.4 (as well as in previous deliverable D2.1, Section 2.5). Beside the architectural design and the technological choices made, we describe the implementation that has been made available to the project partners and is already exploited within WP2 and WP4 activities.

Section 4.6 describes the HPC2I approach toward AI application deployment and management, compares the overall approach with state of the art, closely related ones. The section outlines the design choices for the application support, as well as the implementation tools and technologies that are being used in the development.

Section 4.7 discusses the extension of the METrICS instrumentation and monitoring tool within WP2 in order to support new types of boards, Operating Systems (real-time vs. non-real-time multitasking) and data gathering protocols (offline vs. online). This activity is partly done within the use cases, as a customization to specific hardware, and partly in this WP within task T2.2, as it affects the general usability of the tool within the whole HPC2I architecture.

Section 4.8 presents the status and results of the demonstrator for Task T2.7, that focused on producing an AI-based CPS for the real time autonomous road analysis of a small model car, exploiting tools that will allow moving from a software implementation on the embedded computing device, to an FPGA based one, and towards an ASIC-based implementation.

4.1 Technological Choices and Tools Update

While most of the technological choices made in WP2 activities are discussed in the corresponding subsections, a few choices and changes are discussed here as they are more general in nature. Some changes in the technological choices were mandatory as contingency solution that allowed the project to progress.

The first version of the HPC2I platform is being tested using NVIDIA Jetson Nano boards as edge devices. While in the project plan there is still the aim of porting the software stack on real-time edge boards, due to the global chip shortage the boards supply from I&M was late and still does not cover the whole set of partners. The move to a more industry standard edge hardware has helped minimize the delay in development and allowed producing actually working demonstrators. The effort of porting to a custom combination of HW and SW is necessarily pushed to the second half of the project

Clearly, the choice of testing the AI portion of the CPS over standard devices without a full real time OS has meant that a different kind of separation is enforced among the dependable part of the CPS and the non-dependable one. This issue was well managed, as the requirements on the collaboration among the dependable and non-dependable portions were actually formulated right from the start in a way that allowed some form of decoupling (e.g., requirements *ID_2* to *ID_8*). The two execution domains may possibly even be provided by two separate hardware devices (e.g., two separate boards, or cores on the same CPU that are statically pinned to the two domains by the OS or the hypervisor).

4.1.1 Requirements On Mission-Critical Dependable Subsystem

Two of the requirements related to the dependable subsystem of the CPS were updated.

- Requirement ID_8 is actually relevant for WP5 activities, it was a typo to state that a common core clock with that precision is relevant to all WPs.
- Requirement ID_14 is currently not fulfilled, as the platform in use supports yocto Linux, and not a real time OS.

4.2 Integration Status

HPC2I, the system under development in the context of the WP2 is, as a matter of fact, composed by a set of different modules, each working at a different level and providing, overall, a very sophisticated solution for supporting the TEACHING vision. The complexity of the solution also naturally reflects on the integration. Besides offering different services, each module has its own requirements in terms of base technologies, on top of which it develops to bring to the system peculiar features contributing to the realization of HPC2I. The heterogeneity characterizing such technologies calls for a very careful management of the integration, both in terms of technologies and procedures.

4.2.1 Revised integration plan

The status of the integration in the WP2 is slightly behind the original schedule (this aspect has also been reported in the periodic project report). Such a delay has been caused mainly by the limitation caused by the pandemic, which prevented the opportunity of conducting the planned ad-hoc face-to-face meetings focused on system integration. Such kind of events are

particularly efficient when in presence and quite more complex when conducted adopting a remote setup.

However, several correction actions have been taken into consideration in the context of WP2 to mitigate the impact of the travel limitations on the integration process:

i) A first action concerns with strengthening the connection and the interaction of the integration strategy of the project, managed by WP1. In this perspective the plan is to start a series of periodic integration calls to be jointly organized with WP1. This would help to align the abstract concept and design provided by WP1 with the actual implementation of the WP2.
ii) As a second action, we plan to transform the original integration plan into a live document. This document can be contributed by any WP2 participants and provide a reference point for the status of the integration. The intention is to keep it up to date in pace with the actual development of the platform, so to serve also as a reference point for the partners outside WP2.
iii) The third action for the HPC2I integration strategy refers to the setup of a ticketing system to keep track of the issues hindering the integration strategy. CNR plan to set up a Redmine instance aimed at this goal.

In case the pandemic status will be under control in a way such that travelling will be a concrete option, the integration plan will be amended to consider such a possibility.

4.2.2 Integration actions conducted so far

Beyond the planned action mentioned above, that are aimed at realizing the integration process targeting the complete HPC2I, some actions have been already conducted to partial integration of the modules under development in WP2 with the rest of the project. There are two main chapters of this partial integration that are already providing results so far.

AIaaS Learning modules. The Learning Modules (LMs) under development in WP4 adopt a structure and patterns of (mutual) interaction and interconnection that have been conceived and designed to take advantage of the functionalities provided by HPC2I.

Following the recent trends and best practices, the LMs will be packed as **microservices** (most likely Docker, but this choice might need to be reconsidered depending on the specific tools availability on Yocto Linux). The microservices will be eventually deployed and executed on top of the HPC2I. By using a microservices-based execution, LMs providers can define high-level directives to control the runtime behavior of the application. These directives are enforced by the HPC2I runtime, which drives the placement and deployment of the microservices.

Information Flow. The WP2 agreed to an interconnection network pipeline based on MQTT and Kafka (depending on the actual destination and/or source of the information exchanged). This choice has been taken because of the integration endeavor contributed by UNIPI and CNR that being located on the very same city had the opportunity to organize joint integration session in presence. The network will be also used to connect the LMs of the above mentioned AIaaS.

MetriCS integration. The METrICS framework is used in TEACHING for runtime resource usage measurement of safety critical components. In the context of WP2, the framework has been extended with additional sampling modes and compatibility with non-real-time Linux OSes. In the context of WP5 the framework has also been ported by TRT on the boards developed by I&M, enabling the sampling of the board specific hardware counters. For this aspect we refer the reader to D5.2.

In-Silicon AI. The main research activity in the context of task T2.7 (*Silicon-born dependable AI*) was initially focused on NN-based AI algorithms for vehicle driving analysing radar sensor data. The activity (see Section 4.8) developed a test platform that exploits a small model vehicle on a miniature track and sensor data processing via a small, embedded computer. Such design choices allow real world experiments and data gathering with minimal expenses (and regardless of Covid-related social distance regulations). Results so far are promising, and while the task initially was not deeply integrated with other WP2 activities, a series of meeting since M17 has devised the following integration path

- Online integration of the data sampling on the vehicle with the HPC2I networking base on MQTT/Kafka
- Generation of sample datasets that will be exploited within the project and may also be released under FAIR principles.
- Transposition of the AI prototype to an ASIC-based computing board designed and provided by IFAG, which sports a custom tensor computation accelerator.
- Comparison of the AI methodology exploited in the experiments with the LM developed in WP4, to assess the viability of directly exploiting the same board or a similar one for other TEACHING applications.
- As the monitoring of the vehicle surroundings can be presented to a human subject, human-in-the-loop models can be also trained with this additional source of information.

4.3 High-Level Processing of Data Streams

The goal of task T2.3 is to provide a streaming library capable of implementing the pre-processing stages of the data processing pipeline to be supported in the TEACHING ecosystem. Let us consider, for example, the automotive use case. In this scenario, raw measurements (e.g., speed, position, human status information) coming from different cars running on the monitored area (e.g., a Smart City) are continuously received by localized road-side units that must be equipped with sufficiently powerful, but still low power, computational resources. There, several pre-processing stages must be applied to prepare the retrieved data for the next main computational steps, such us high-speed inference processes to be computed on the fly by transferring back notifications or personalization results to the running cars. Typical pre-processing stages include sampling, data reduction through aggregates applied on sliding windows, basic streaming transformations, joins and so forth. In this section, we provide a concise summary of the current results provided in T2.3.

4.3.1 Streaming Patterns and Provided Operators

The main outcome of T2.3 is to enhance the WindFlow[43] streaming library with a flexible and extendible set of streaming operators to be easily included in streaming applications runnable on multicores. WindFlow is a C++ header-only library, designed by using generic programming leveraging the recent features of the C++17 standard for template argument deduction.

At the current stage of the project, WindFlow provides a set of *basic* and *windowed operators* that can be interconnected in dataflow streaming graphs called *topologies*. Operators can be internally replicated to increase their throughput, with internal replicas working concurrently on in parallel on a subset of the inputs received by the whole operator. Table 2 shows the currently supported operators. The column *distribution* indicates how the inputs are delivered to the operator replicas: *forward* means that every input can be assigned to any replica (randomly or in a load-balanced manner), *keyby* sends all the inputs having the same key attribute (e.g., a specific field of the input records) to the same replica, and *complex* distributions deliver the same input to one or more replicas according to more sophisticated policies that are operator dependent. For the Source operator the distribution feature does not apply since Source replicas never receive any inputs.

Table 2: Set of supported operators in WindFlow

	Operator	Acronym	Distribution
Basic	Source	SRC	-
	Sink	SNK	Forward/KeyBy
	Filter	F	Forward/KeyBy
	Map	M	Forward/KeyBy
	FlatMap	FM	Forward/KeyBy
	Accumulator	A	KeyBy
Windowed	Keyed Windows	KW	KeyBy
	Parallel Windows	PW	Complex
	Paned Windows	PAW	Complex
	Map-Reduce Windows	MRW	Complex

The *Source* operator is in charge of generating a stream of data as inputs for other operators (e.g., records of attributes like a sensor reading, or unstructured data like a text or an image). The *Sink* operator is responsible for absorbing its inputs without generating any output (e.g., by consolidating them into a database or in a distributed file system). The *Filter* operator drops all the tuples not respecting a user-defined predicate. The *Map* operator produces one output per input while the *FlatMap* operator produces zero, one or more outputs per input (inputs and outputs may have different data types). The *Accumulator* operator maintains an internal state for each value of the key attribute. For each input, a user-defined processing function works on that input and on the corresponding state of its key, and the new value of the state is produced in output by the operator (and kept internally updated).

[43] https://github.com/ParaGroup/WindFlow

In addition to basic operators, which are general set of transformations for building streaming topologies, we have included a rich support to windowed operators capable of repeating a user-defined processing function over the most recent data grouped into a so-called *window*. Different models have been presented in the literature. The *triggering semantics* expresses when a window is ready to be processed while the *eviction semantics* states which are the data items that must be purged from the window after its triggering (because they are too old and not meaningful anymore). Common semantics use the number of items (*count based*) or a timestamp attribute in the items (*time based*). As an example, time-based windows trigger every $s > 0$ time units (sliding factor) where each window spans over the last $w > 0$ time units (window length).

Window-based computations are often time consuming, because when the input stream comes with a high speed, consecutive windows can trigger very frequently by increasing the computation burden. Therefore, also on embedded devices, the implementation of such operators needs to be carefully designed to leverage multicores and accelerators like FPGAs and GPUs. In the current stage of the work, WindFlow provides parallel implementations leveraging multicores and capable of expressing parallelism in different ways:

1. *inter-key parallelism*: this is basically the most common pattern, where replicas of the same operator process triggered windows of different keys in parallel.
2. *inter-window parallelism*: consecutive triggered windows even with the same key attribute can be assigned and processed in parallel by distinct replicas of the same operator.
3. *intra-window parallelism*: when specific properties of the processing function are known (e.g., associativity and commutativity), the processing of the same window can be executed in parallel by the replicas of the operator.

These parallelism strategies are sketched in Figure 7 to give a graphical representation of them.

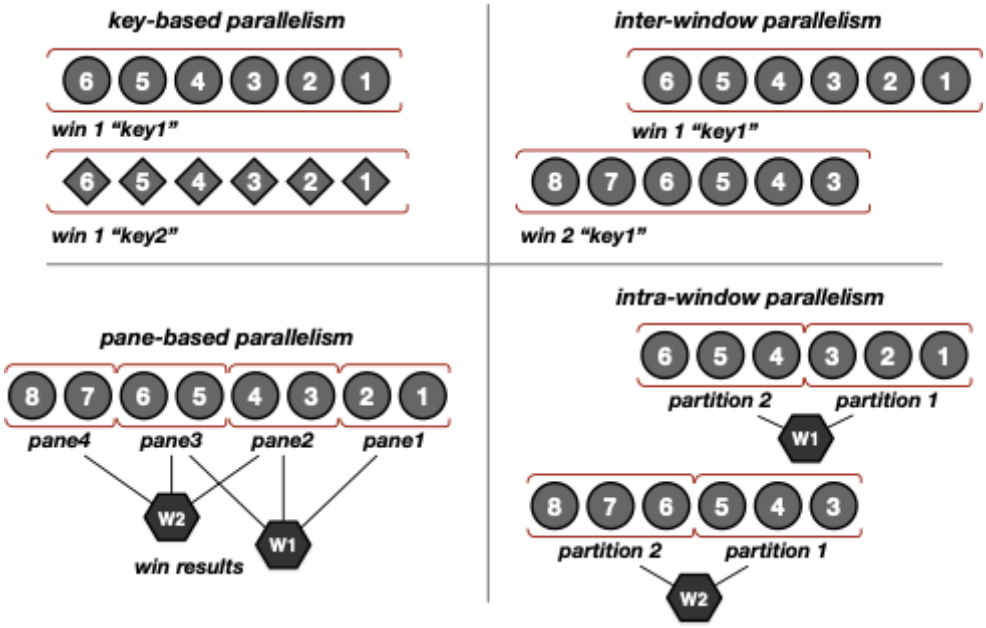


Figure 7: Parallelism strategies for windowed operators.

Such parallel strategies are provided to the final user according to the *parallel design patterns* methodology[44]. In other words, each strategy is provided to the user as a separate operator having a clear API that hides all the internal implementation aspects related to how inputs are partitioned among the internal replicas, how partial results are managed and processed to produce final window-wise results, and so forth. Table 3 shows what are the high-level operators for window-based processing, and which parallelism model is adopted by each of them.

Table 3: Window-based parallel patterns in WindFlow and their parallelization strategies

	Parallel Windows	Keyed Windows	Paned Windows	MapReduce Windows
Inter-key Parallelism	✗	✓	✗	✗
Inter-window Parallelism	✓	✗	✓	✗
Intra-window Parallelism	✗	✗	✓	✓

The *Paned Windows* pattern is a parallelization of the approach originally described in a paper from Li *et al*[45]. The approach avoids recomputing windows from scratch by exploiting the partial overlapping between two consecutive windows. It is based on a two-level aggregation that uses the notion of *pane*: panes are non-overlapping windows of length equal to $p = GCD(w, s)$. The length can be expressed in number of items or in time units depending on the model (count or time). The computation consists of two phases: the *Pane-level Sub- Query* (PLQ) computes a result for each pane, while in the *Window-level Sub-Query* (WLQ) the results of the w/p panes belonging to the same window are merged to produce the corresponding window result. Since panes can be computed in parallel (shared by the same windows or even belonging to different windows), this pattern can support both inter-window parallelism and intra-window parallelism. The *MapReduce Window* pattern instead, since it partitions each window from scratch, is able to support intra-window parallelism only, and partition results are not reused for computing the final results of more distinct windows.

In the next part, we will focus on the high-level API provided by WindFlow to express basic and windowed operators, and we explain how streaming topologies can be built and run on multicores.

4.3.2 WindFlow API

The API of WindFlow has been designed with specific principles in mind. In particular, it tries to be similar to the ones of traditional streaming frameworks for clusters and clouds, such as Apache Storm[46] and Flink[47]. Our goal is to target general-purpose stream processing, with support for generic operators as well as their stateful definition (with user-defined states provided in input to the operator business logic code). In the description of the API, we first

[44] T. Mattson, B. Sanders, and B. Massingill, Patterns for Parallel Programming, 1st ed. Addison-Wesley Professional, 2004.

[45] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No Pane, No Gain: Efficient Evaluation of Sliding-window Aggregates over Data Streams," SIGMOD Rec., vol. 34, no. 1, pp. 39–44, Mar. 2005.

[46] https://storm.apache.org/

[47] https://flink.apache.org/

focus on how operators can be created and configured in the library. Then, we will see how operators can be interconnected in acyclic graph topologies.

WindFlow provides a compositional fluent interface based on *builder* classes. Figure 8 shows how to create a Map using its builder class `Map_Builder`. By leveraging the recent features of C++17, the template arguments for instantiating the Map class (the data types `input_t` and `output_t` in the figure) are automatically deduced by the signature of the function provided to the builder constructor.

```
class Map_Function {
public:
    output_t operator()(const input_t &t) {
        … // code here
    }
};
Map_Function mapF;
Map map =
Map_Builder(mapF).withName("myMap").withParallelism(5).build();
```

Figure 8: WindFlow API to create a Map operator

The library does not rely on class inheritance to define the logic of the operators (to avoid the little overhead of virtual function calls). WindFlow accepts several signatures for each operator, where the logic can be provided either as a plain function, as an anonymous lambda, or through functor objects like in Figure 8. This latter option allows logics that are not purely functional to be used by the operator: the state can be implemented as internal variables of the functor object, and the runtime system guarantees that each replica uses a distinct copy of the object.

The customization is done with the method chaining technique, where configuration options are set with specific methods. Finally, the `build()` method creates the instance of the properly configured operator. In the figure, the Map is created with five replicas and with a name (the string "*myMap*") used for logging purposes. Other options are available depending on the operator type.

A recent feature that we have added to version 3.0 of WindFlow, developed entirely during the activities of T2.3, has been to provide a nano-batching support (buffering of few inputs) in order to improve throughput on multicores, by amortizing the run-time overheads. This feature can be applied on a per-operator basis during the customization of the operator with its builder. The builder method `withOutputBatchSize(value)` can be used to indicate the buffing applied in output by the operator. If no batch size is provided (or if it is zero), each output is delivered to the next operator as soon as it is available. Otherwise, messages are delivered in small batches whose size can be chosen with a fine granularity by the user.

Applications are developed using the *PipeGraph* and the *MultiPipe* constructs. The PipeGraph is the environment where a streaming application is created. It can be configured with some global properties related to how the overall streaming computation must be performed. It supports three *execution modes*:

1. ***deterministic***: in this mode the user is requested to generate inputs in the Sources ordered by timestamps. The run-time environment guarantees that all the replicas of the operators will process inputs ordered by timestamp.
2. ***probabilistic***: in this mode the Source replicas generate outputs in any order, but the run-time system still sorts items before processing the functional logics of the operators. In this case, to respect the ordering, items can be dropped.

3. ***default***: this is the mode without ordering guarantees. Inputs are processed in a non-deterministic ordering and window-based computations are evaluated based on watermark propagation.

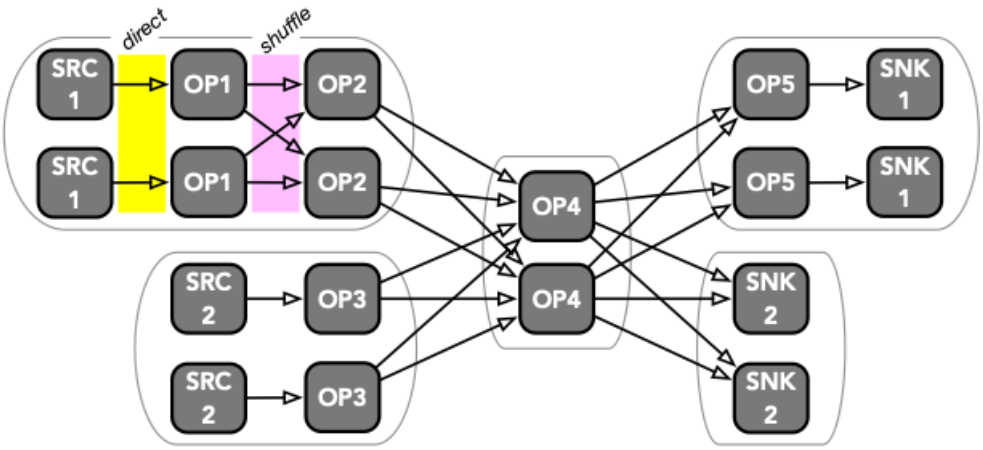
The basic compositional block to build a WindFlow application is the MultiPipe. A MultiPipe can be created by adding a Source to the PipeGraph. Figure 9 shows an example of instantiation, where the MultiPipe is fed by a Source (previously created, and not shown in the code snippet for brevity). Then, two previously created operators are added to the MultiPipe. Finally, a Sink is added to the MultiPipe. The application is run by invoking the `run()` method on the PipeGraph. It terminates when all the Source replicas terminate, and all the generated values have been fully processed.

```
PipeGraph app("myApp");
MultiPipe &mp = app.add_source(mySource)
                   .add(myFilter).add(myMap).add_sink(mySink);
app.run();
```

Figure 9: Example of a streaming application consisting in one MultiPipe.

In terms of internal interconnections, a *MultiPipe* is a set of parallel pipelines of operator replicas, where a replica of an operator in a pipeline communicates either with one or with all the replicas of the next operator. When a replica receives inputs only from one replica of the previous operator (in the same pipeline), we call this communication pattern a *direct connection*. A *shuffle connection* is when a replica receives inputs from all the replicas of the previous operator (e.g., because tuples are distributed on a *keyby* basis).

Furthermore, to create more complex graph topologies, the API provides two methods named `split()` and `merge()`. The output stream generated by the last operator in a MultiPipe can be split to feed different MultiPipes, each starting with a different operator. The API provides the user with the possibility to provide a *splitting logic* (e.g., though a lambda) to indicate how the output items must be delivered to the different destinations (e.g., unicast, multicast or broadcast). The merge method allows the output streams of different MultiPipes to be multiplexed by feeding a destination MultiPipe. Figure 10 shows a more complex example of how the API can be used to create a more complex topology. In the figure, for the sake of simplicity, we suppose that all the operators have been previously created (through their builder classes) with two internal replicas. The figure shows both the API and the resulting interconnection of MultiPipe structures.




```
PipeGraph app("myApp");
MultiPipe &mp1 = app.add_source(SRC1).add(OP1).add(OP2);
MultiPipe &mp2 = app.add_source(SRC2).add(OP3);
MultiPipe &mp3 = mp1.merge(mp2); // merge of mp1 and mp2 into mp3
mp3.add(OP4).split(
      [](const tuple_t &t) -> std::vector<int> {
         ... // code returning either 0 or 1 or both per input
      }, 2);
mp3.select(0).add(OP5).add_sink(SNK1);
mp3.select(1).add_sink(SNK2);
app.run();
```

Figure 10: Example of a more complex application in WindFlow. The use of the API and the resulting application structure are shown.

Split and merge of MultiPipes are operations that requires the enforcement of specific type constraints. As an example, the output streams of different MultiPipes can be merged if and only if all the output items have the same data type. Similar constraints must be enforced when a new operator is added to a MultiPipe and in the split of an existing MultiPipe. These checks are done in the `run()` method by using the RTTI (RunTime Type Identification) feature of C++. In the next section, we present the formal approach that we used to design the runtime system.

4.3.3 Structured Runtime

The implementation of the WindFlow run-time system follows a novel design approach based on the combination of efficient concurrent building blocks, which are particularly optimized for multi-core architectures. In terms of layers, WindFlow leverages the building block implementation provided by the FastFlow[48] parallel programming framework, which is an open-source library for parallel computing developed at the University of Pisa since 2009. Figure 11 shows the layered structure of the FastFlow ecosystem.

[48] https://github.com/fastflow

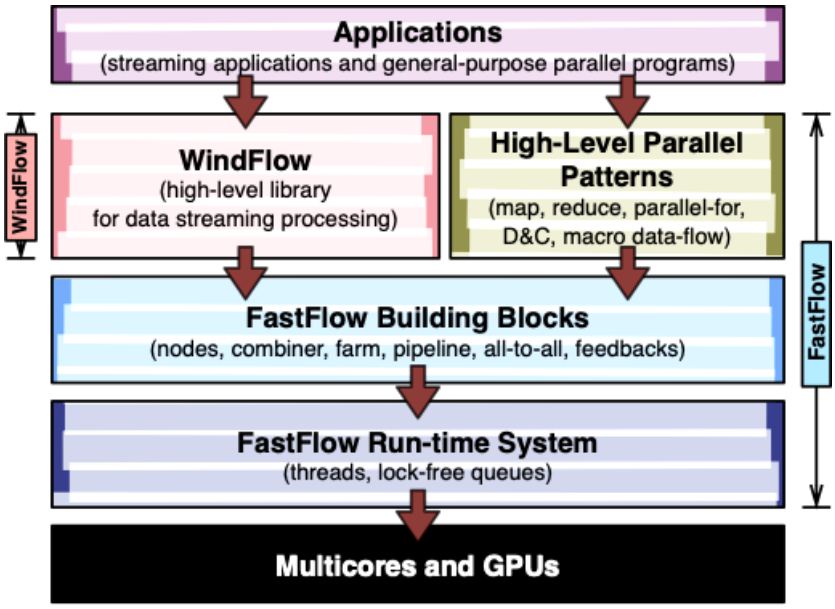


Figure 11: Abstraction layers of the FastFlow ecosystem.

While the high-level API allows user to prototype streaming applications in a easy manner, the internal implementation of streaming topologies is obtained by a formal structuring of few building blocks summarized as follows:

1. ***Nodes*** are the unit of sequential execution. They encapsulate either user code or run-time system code. A generic node performs a loop that: *i)* gets a data item (through a memory reference to a data structure) from one of its input queues; *ii)* executes a functional code working on the data item and possibly on a state maintained by the node itself by calling its service method `svc()`; *iii)* puts a memory reference to the resulting item(s) into one or multiple output queues selected according to a predefined or user-defined policy. By default, each node is executed by a dedicated run-time system thread executing continuously its internal loop.
2. ***Combiners*** allow the user to combine two nodes into one single sequential node. Conceptually, the operation of combining sequential nodes is similar to the composition of two functions. This block can also be used to reduce the threads used to run the data-flow network by executing the functions of multiple nodes by a single thread.
3. ***Pipeline*** allows more building blocks to be connected in a linear chain. It is used both as a container of building blocks as well as an application topology builder. At execution time, the pipeline building block models the data-flow execution of its building blocks on data elements flowing in a streamed fashion.
4. ***All-to-All*** (A2A) defines two distinct sets of blocks connected as in a full crossbar. This means that each block in the first set (called left set) is connected to all the blocks in the second set (called right set). Although the topological shape is a full crossbar, the user can implement any custom distribution in the left blocks (e.g., sending each data item to a specific block in the right set, shuffling or broadcasting).

Blocks can be combined according to specific composition rules. For example, blocks in the two sets of a A2A can be nodes, combiners, pipelines or even other A2A blocks. In the same way, blocks in a pipeline can be other pipelines, A2A, or sequential blocks (nodes or combiners).

One distinguishable aspect of the FastFlow's building blocks is their efficient implementation. This is mainly related to the design of the queues used to exchange data pointers between nodes, which are efficient lock-free single-producer single-consumer queues with latencies of few hundreds of nanoseconds on commodity multicores[49]. Such queues can be configured to use a predefined capacity (bounded queues), or rather to be unbounded. In the former case, limited-size queues allow a simple yet effective backpressure mechanism to be implemented for streaming topologies, by keeping the overall memory occupation stable during runtime (feature particularly important for small devices with limited memory availability).

The way in which building blocks are composed to generate streaming topologies resulting from the high-level API is a complex aspect of the WindFlow design, which has been described in terms of a formal semantics in a publication[50] that is one of the most meaningful scientific result of the activities conducted in T2.3. Here we provide for the sake of completeness a general idea of how such blocks are efficiently composed.

Let us consider the code fragment in Figure 12, showing the use of the high-level API to instantiate a streaming topology with a Source connected to a Map. Outputs from the Map are processed by a Filter operator by eventually sending retained items to the Sink. Each operator is instantiated with two internal replicas, implemented by FastFlow nodes (and therefore by dedicated threads).

High-level C++17 API

```
PipeGraph app;
Source src = Source_Builder(…)
                  .withParallelism(2)
                  .build();
MultiPipe &mp = app.add_source(src);


Map map = Map_Builder
               ([](input_t &t) -> void {…})
              .withParallelism(2)
              .build();
mp.add(map);

Filter filter = Filter_Builder
                  ([](const input_t &t) -> bool {…})
                  .withParallelism(2)
                  .withKeyBy
                   ([](const input_t &t) -> key_t {…})
                  .build();
mp.add(filter);

Sink sink = Sink_Builder(…)
                 .withParallelism(2)
                 .build();
mp.add(sink);
```

Figure 12: Code fragment of a simple streaming application in WindFlow.

[49] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in Euro-Par 2012 Parallel Processing, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 662–673.

[50] G. Mencagli, M. Torquati, A. Cardaci, A. Fais, L. Rinaldi and M. Danelutto, "WindFlow: High-Speed Continuous Stream Processing With Parallel Building Blocks," in IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 11, pp. 2748-2763, 1 Nov. 2021, doi: 10.1109/TPDS.2021.3073970.

This high-level code is translated into a composition of building blocks like in Figure 13. The overall application structure is one A2A block with two pipelines on the left-hand side and other two pipelines in the right-hand side. Each Source replica is directly connected to one replica of the subsequent Map. This happens because two constraints are satisfied: *i)* Source and Map have the same number of replicas; *ii)* the Map accepts inputs from the Source without a *keyby* distribution. The *keyby* distribution is instead utilized for the Filter operator, which is created by providing a key extractor logic to extract the key attribute from each input item. Since the Map replicas shall send all the outputs with the same key to the same Filter instance, a shuffle communication pattern is used, implemented through a A2A block. It is worth nothing the use of combiners for fusing in the Map replicas the routing logic to distribute outputs from the Map properly to the Filter replicas (nodes marked with "E" in the figure). The same idea is applied to the collection nodes "C" combined with the Filter replicas. They might implement different collection strategies depending on the execution mode of the whole streaming environment (PipeGraph). As an example, in case of a PipeGraph created in DETERMINISTIC mode, the collection nodes "C" are in charge of reordering inputs based on their timestamps, before applying the filtering logic on them.

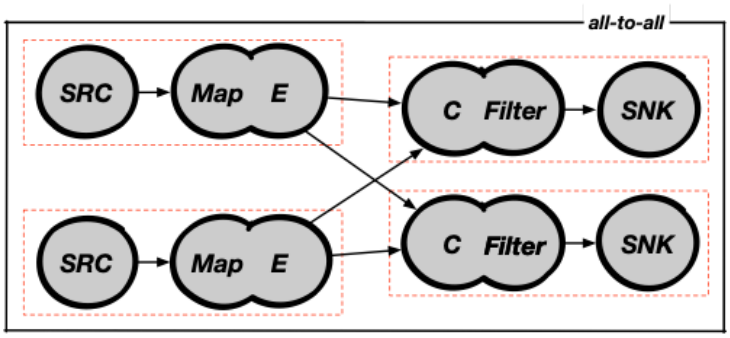


Figure 13: Implementation of the topology depicted in Figure 12 via FastFlow building blocks

A more complete overview of the WindFlow run-time system can be found in a paper from Mencagli et al.[51]

4.3.4 Evaluation Results

In this phase of the T2.3 activities, we have performed a preliminary evaluation in order to compare the performance provided by WindFlow against the ones of traditional streaming frameworks, in particular Apache Storm and Flink. This analysis is useful to understand the potential of WindFlow in providing a novel streaming support for single machines, without the overheads traditionally affecting the design of more complex frameworks designed to target distributed systems like clusters and clouds.

The evaluation has been performed in a server machine equipped with two AMD EPYC CPUs, each featuring 24 cores for a total of 48 cores working at 2.4 GHz. For the comparison we use Storm version 2.1.0 and Flink version 1.9.0. Furthermore, we compare the performance of WindFlow against the one provided by a recent research prototype for streaming on single

[51] G. Mencagli, M. Torquati, A. Cardaci, A. Fais, L. Rinaldi and M. Danelutto, "WindFlow: High-Speed Continuous Stream Processing With Parallel Building Blocks," in IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 11, pp. 2748-2763, 1 Nov. 2021, doi: 10.1109/TPDS.2021.3073970.

machines (BriskStream[52]). All the compared frameworks adopt the JVM in the runtime system, while WindFlow is completely written in C++. So, the performance advantages are both related to the choice of a faster language and to the novel design of the run-time system that we developed.

Performance results are collected using a benchmark suite of 7 streaming applications part of DSPBench[53], a complete suite of streaming applications that we developed during the T2.3 activites of the project. The applications are FraudDetection (**FD**), SpikeDetection (**SD**), TrafficMonitoring (**TM**), WordCount (**WC**), Yahoo Streaming Benchmark (**YB**), LinearRoad (**LR**), and VoipStream (**VS**). They come from different domains: finance (FD, SD), smart cities (TM, LR), synthetic (WC), advertisement (YB), telemarketing analysis (VS). They represent interesting use cases where fast input streams must be processed on the fly to extract insights and alterts, which is a typical use case of the TEACHING project.

Performance is evaluated in terms of throughput and latency. Throughput indicates the maximum number of inputs that the streaming application is able to process per time unit. Results are shown in Figure 14. The first two subfigures (a) and (b) show the results without parallelizing the operators within the considered topologies. On average, WinFlow is 11.5, 9.2 and 9.8 times faster than Storm, Flink and BriskStream. The largest improvement is with TM, where it also depends on the differences between the two libraries used for geolocalization (we used GDAL instead of the original GEOTOOLS which is not available in C++). The second two plots (c) and (d) show the results with a parallelization of each operator using the best configuration found for each framework. Also in this case, WindFlow outperforms the others by achieving a highest throughput in all the considered applications.

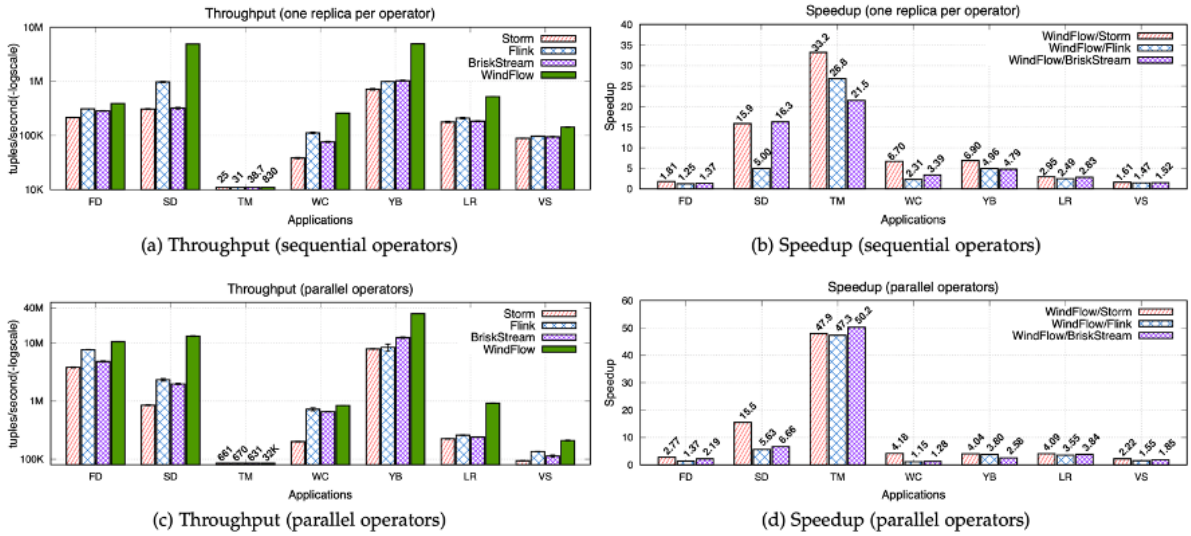


Figure 14: Throughput comparison, WindFlow against Apache Storm, Flink, and BriskStream

[52] Shuhao Zhang, Jiong He, Amelie Chi Zhou, and Bingsheng He. 2019. BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures. In Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19). Association for Computing Machinery, New York, NY, USA, 705–722. DOI:https://doi.org/10.1145/3299869.3300067.

[53] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer and L. G. L. Fernandes, "DSPBench: A Suite of Benchmark Applications for Distributed Data Stream Processing Systems," in IEEE Access, vol. 8, pp. 222900-222917, 2020, doi: 10.1109/ACCESS.2020.3043948.

Streaming applications are characterized by strict real-time latency constraints to work in an effective online manner. Latency refers to the elapsed time from the reception of an input item by the Sources to the time instant when the system completes the processing of the input and the corresponding result in the Sink has been produced to the end users. Figure 15 shows an analysis of the latency exhibited by WindFlow against its competitors by considering for brevity a subset of the applications utilized before.

The latency is affected by the specific features of the streaming frameworks, like the size of the message queues. To have a fair comparison, the applications have been run with a controlled input rate of 10K tuples/second, and we fixed the size of the message queues to 32K entries. The results show that WindFlow provides small and stable latency values. The latency provided by BriskStream is very similar to the one of Storm (except in FD and VS). Among the traditional systems, Flink provides the lowest latency. On average, the mean latency obtained by WindFlow is 12.7, 8.49 and 9.28 times smaller than the one of Storm, Flink and BriskStream. Furthermore, we point out that WindFlow can achieve a significantly lower tail latency.

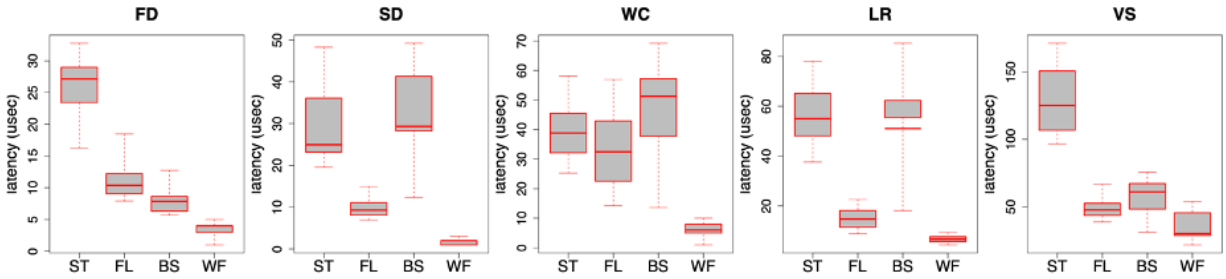


Figure 15: Latency comparison among frameworks. WindFlow (WF) against Storm (ST), Flink (FL) and BriskStream (BS).

4.3.5 Overview of Task Activities and Plans

In conclusion, the activities conducted by T2.3 during this first half of the project have been focused on the definition of a streaming library that is particularly effective and efficient for multi-core architectures. Stream processing is a sufficiently general computing paradigm, that can be adopted in different phases of the TEACHING processing pipelines in a CPSoS. For pre-processing stages of raw sensor data, stream processing can be used to enrich raw inputs with insights and useful information required for the next stages of the process of learning and inference. Fast inference can be integrated in WindFlow topologies as special purpose operators, which will be the focus of the next steps of task T2.3 activites. In terms of integration with the other WP2 tasks, incoming data arriving at a WindFlow applications, and the outgoing results, can be easily integrated with messaging technologies like Apache Kafka and MQTT. The definition of proper Source and Sink operators will be conducted among the next activities of task T2.3.

4.4 Exploiting Accelerator Devices

The goal of task T2.2 is to build a set of abstractions to ease the use of accelerators for streaming tasks and for interfacing with the other AI computations developed in WP4. Two kinds of accelerators are considered for this goal: integrated GPU devices available as System-on-Chip boards, and FPGA-based programmable devices. The effort conducted in the first part of the project for task T2.2 has been related to the development of a run-time support for streaming on integrated-GPUs while the activity targeting FPGA-based boards is still under preliminary study.

Pilot Device: NVIDIA Jetson Nano

The target board used for the project task is the NVIDIA Jetson Nano board. The board is a small low-powered device shipped by NVIDIA. It is equipped with an ARMv8 4-cored CPU (64 bit) working at 1.43 GHz, equipped with 4GB of RAM and one integrated GPU composed of a Stream MultiProcessor (Maxwell) with 128 CUDA cores. The GPU shares the same physical 4GB memory of the CPU.

Compared with traditional "discrete GPU" models (i.e., the ones available as PCI-ex extension board of traditional hosts), the NVIDIA Jetson Nano exhibits all the features and characteristics of a device perfectly mimics smart Edge-based solution for in-situ computing of data coming from sensors and other data providers. The activity related to task T2.2 has been conducted with the goal of extending the streaming library developed in T2.3 with low-level mechanisms abstracting the interaction with a GPU device in an efficient and user-friendly manner.

4.4.1 GPU-based Streaming Operators

The outcome of the activity of T2.2 has been conducted in parallel with the streaming support developed separately in T2.3. While T2.3 is devoted to identifying and implementing useful streaming patterns on multi-core architectures, T2.2 is aimed at extending the streaming support to seamlessly interfacing with various accelerators available in the target platforms and devices envisioned in the TEACHING technological ecosystem.

From the high-level perspective, the WindFlow library for parallel stream processing has been recently extended to include two operators whose processing can be easily offloaded on a GPU device like the integrated GPU available on the Jetson Nano board:

1) ***Map operators***: map operators are computations that traditionally fit the SIMD capabilities provided by GPU devices. A map operator is instantiated in the library with a user-defined function applied on each received input and producing an output result. In general, the precondition is that all the inputs can be processed in parallel. The offloading on GPU consists of buffering a set (batch) of continuous inputs, launching a CUDA kernel capable of executing in parallel all the inputs in the buffer, and producing and output batch of results.
2) ***Filter operators***: filter operators apply a user-defined predicate on each input returning a Boolean value. All the inputs whose predicate is evaluated to *true* are produced in output from the filter operator, while all the inputs with predicate evaluated to *false* are dropped and discarded from the further steps of the streaming analysis. GPU-based Filter implementations should buffer the inputs in small batches. Each batch, once it is filled, is evaluated in parallel by the GPU, that is by computing the predicate in parallel for all the inputs. After this step, the batch is cleaned up by removing all the inputs to be dropped.

Although the previous ideas look straightforward, stream processing applications usually adopt more complex semantics in map and filter operators, which are not always stateless as previously indicated. In other words, there are computational patterns in which the result of the predicate computed on each input (for example for a filter) does not depend only on the properties and information in the current input but on the history of the inputs received so far (and possibly on the history of inputs having similar properties with the current evaluated one).

Although many different kinds of stateful computations can be defined in the streaming domain, a very popular pattern has been early recognized by the scientific literature and is often referred to as ***key partitioning***[54]. The general idea can be summarized in three different points:

- Each input item is a data structure (e.g., a record of attribute) including an information called key attribute. It can be a primitive type (e.g., an integer corresponding to a unique identifier) or any type (e.g., a string, a structure and so forth). Each input is associated with exactly one key value.
- Each key value is associated with an internal state, that is managed and updated independently from the state associated with other key values.
- The processing on an input x belonging to key k in order to produce the output value $\underline{y}$ requires reading the input value x and accessing (and possibly updating) the internal state associated with key k.

Such kind of paradigm allows the system to process in parallel inputs of different key values, for which the internal states are different and can be safely modified in parallel. Inputs of the same key shall be executed in arrival order, to allow accessing the state correctly without changing the computation semantics.

Such concept of stateful key-based processing has been already integrated in the WindFlow library for traditional operators executed on the multi-core CPU. Indeed, different threads are responsible from processing inputs, while the distribution entities are responsible to send all the inputs with the same key attribute to the same worker thread in charge of computing them one by one.

In order to implement such computational semantics on GPU, we should observe that inputs of the same batch cannot be safely executed in parallel by independent CUDA threads, but they need to be processed by looking at their key attribute. A set of CUDA threads in the kernel are created and associated to a subset of the key attributes present in the current batch. Each CUDA thread executes sequentially all the inputs of its assigned keys while other CUDA threads apply the same computation on different key attributed in parallel.

Implementing this semantics requires special care to obtain good performance on GPU. Figure 16 shows this idea. In the input batch of the figure, all the inputs with key “k1” are identified and must be processed sequentially, one by one, by the same CUDA thread, which keeps an internal state on the device and updates it by applying the user-defined function taking as input the current input item and the state object.

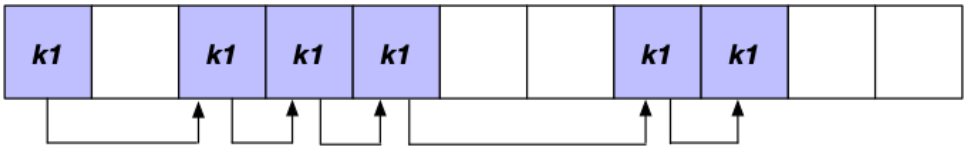


Figure 16: Key-based operator processing in WindFlow.

[54] K. Li, G. Liu and M. Lu, "A Holistic Stream Partitioning Algorithm for Distributed Stream Processing Systems," 2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2019, pp. 202-207, doi: 10.1109/PDCAT46702.2019.00046.

4.4.2 Streaming Run-time Support Leveraging GPUs

WindFlow 3.0.0 is provided with an efficient run-time support leveraging the recent features of CUDA11 together with the C++17 standard. Three aspects are of special importance for the design of the run-time system:

- an efficient implementation of CUDA kernel providing data-parallel execution of map/filter operators on GPU,
- a recycling mechanism to reuse already allocated GPU accessible memory areas,
- specific support for SoC Tegra-based GPUs to support memory prefetching and to avoid unnecessary copy operations. We will focus on these three points in the rest of this section.

CUDA kernels are in charge of computing an available batch of inputs in order to produce the corresponding results. In case of stateless map and filter operators, all the inputs in the batch can be processed in parallel by a distinct CUDA threads, where each thread invokes the user-defined function provided by the user (e.g., through a `__device__` lambda, or function, or functor object with the proper required signature). Although easy to implement, such kind of kernel requires special care. Each CUDA thread is in charge of executing the whole user-defined function (i.e., a map transform or a filter predicate) on its corresponding input. Depending on the nature of this function, thread divergence can easily happen in case of functions with divergent branches. To mitigate such an issue, the assignment between inputs in the batch and corresponding threads in the kernel is done according to the following principles:

1. the kernel is configured to use a maximum number of blocks equal to the maximum number or resident blocks on the GPU device (e.g., on Jetson Nano this number is 32),
2. the number of threads in the kernel is configured to never exceed the maximum number of resident threads on the GPU,
3. depending on the size of the batch, if it is sufficiently small, one thread per warp is configured to process one input of the batch, while the others remain idle,
4. if the size of the batch is greater than the number of resident warps on the GPU, more threads per warp are configured to be actively involved in processing a corresponding input of the batch. In case of very large batches, all the resident threads of the GPU are active and process at least one or more than one (in case of large batches) values of the input batch.

The rationale of the approach informally described by the itemize above is to use only warp if they are sufficient to process in parallel all the inputs of the batch. Indeed, threads of different warps are not affected by thread divergence issues. More threads per warp are instead used if the input batch is very large, to completely fill the occupancy of the GPU.

The same idea is applied to the stateful version of the kernels, where each active thread is assigned to one (or more than one) distinct key attribute, and it is responsible to sequentially process all the inputs of the batch having a key assigned to that CUDA thread. In this process, a state object per key is maintained in CUDA memory and allocated when the first input of that key is received by the system, then it is accessed sequentially by at most one CUDA thread in the kernel (so without consistency and race condition problems in accessing and manipulating it).

Another crucial point for the performance of WindFlow on GPU is to provide a smart recycling mechanisms of memory areas. The library provides different kinds of emitter functionalities used to route input items to destination operators that can be either CPU-based or GPU-based ones, and they can either work on a item-by-item basis (for CPU-based operator only) or in a

batched mode (for both CPU-based and GPU-based operators). Figure 17 summarizes the six possible configurations of the emitter functionality.

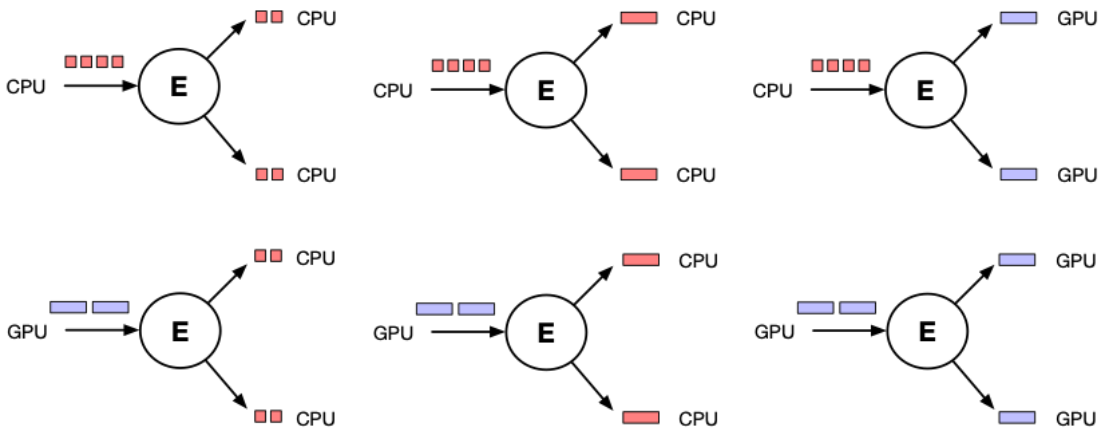


Figure 17: Emitter functionalities of WindFlow 3.0.0.

Such emitters, together with the operator replicas, are properly composed into data-flow graphs representing our streaming applications. On GPU devices, dynamic allocations of GPU-accessible areas (e.g., through cudaMalloc calls) are notoriously expensive, since they produce a global barrier on the GPU devices which waits for the completion of all the previously launched kernels on the device. This problem has been partially mitigated in CUDA11, with the new stream-based allocation support which allows cudaMalloc calls to be bound together within a CUDA stream instead of being issued globally. However, GPU allocation is still expensive and might be detrimental for the performance of the streaming pipeline, where batches are continuously allocated and filled with new items materialized in the stream.

To avoid such a problem, during the activities of T2.2 we implemented a recycling mechanism where already allocated batches can be reused and re-filled with new inputs. This requires a synchronization mechanism to be sure of the time instant when the batch can be re-filled without overwriting useful data not processed yet. This behavior is enabled thanks to the structured run-time system developed for WindFlow, which is essentially based on the building blocks of FastFlow (e.g., farm, all-to-all, combiner and other blocks). Such blocks are used to build acyclic data-flow graphs in a constructive manner, according to a formal compositional semantics. Feedback channels can be used to enable the recycling of the GPU-accessible data structures, since they connect consumer entities of batches with producing ones. The transmission of a batch pointer (or a pointer to a structure containing metadata of the batch) to the producer informs the latter that the consumer did its work on the batch, and the internal arrays can be safely recycled.

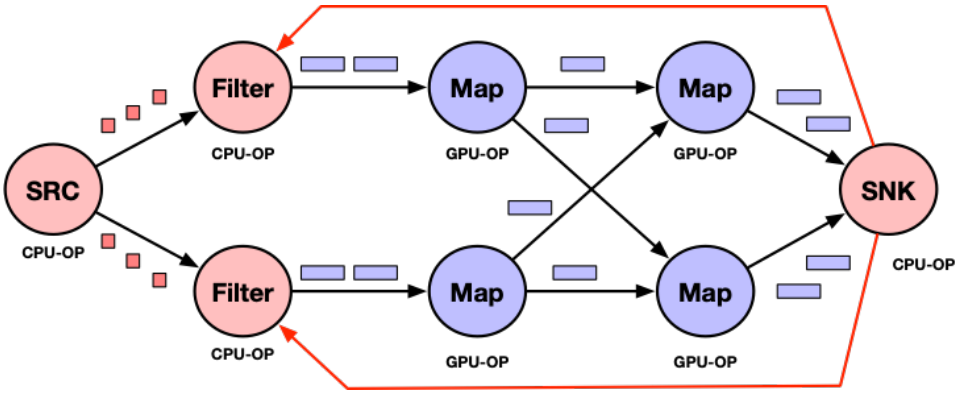


Figure 18: Recycling mechanism of WindFlow.

The abstract behavior described above has been concretely implemented by using multi-producer single-consumer lock-free queues (MPSC), used together to the single-producer single-consumer queues already used by FastFlow for passing pointers to message between nodes of the graph. A MPSC queue is instantiated per producer entity, i.e., a node in the graph responsible to create GPU-accessible batches. The pointer to the queue, together with other metadata information, is added to the batch additional fields. So doing, when the final consumer of the batches consumes their inputs, the internal arrays can be recycled by inserting the batch pointer to the corresponding MPSC queue (so giving back the ownership of the arrays to the producer entity).

In Figure 18 a source operator produces a stream of data items transmitted to a filter operator (except for the source and the sink). The figure shows just two replicas per operators for the sake of simplicity. The two map operators (the first stateless, the second stateful) are executed on a GPU device, while results are collected by the final sink operator.

The emitter functionality connected to the filter replicas is responsible for receiving inputs one by one, and for filling a GPU-accessible batch. It implements a simple protocol that first checks the presence of a previously allocated batch ready to be recycled from its MPSC queue (red arrows in the figure). If there are no ready batches in the queue, a new batch is allocated.

In addition to the aforementioned characteristics of the WindFlow's run-time system, we have studied other optimizations related to:

- avoiding additional memory copies: in discrete GPUs, the use of explicit memory transfers can outperform the use of unified memory, since it avoids useless memory transfers from device to host for internal arrays of batch structures that must be recycled (and so their content is totally overwritten by the host). In integrated architectures like the Jetson Nano, the CPU and the GPU share the same memory and the use of CUDA unified memory revealed more efficient than other approaches to share memory (e.g., like the use of pinned memory, which is not cached on both CPU and GPU sides on Jetson Nano[55]).
- The run-time system has been entirely designed following the producer-consumer paradigm. In addition to the recycling of batches, the run-time system eases the use of CUDA prefetching hint calls, that when properly introduced can improve the

[55] https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html

performance of the CUDA driver and of its activities to make CPU and GPU caches coherent with each other.

- CUDA streams have been efficiently used in the run-time system. Using multiple CUDA streams allow different host threads to launch kernels that can run concurrently or even in parallel on the GPU. In WindFlow, a CUDA stream is created per on-the-fly batch, and the run-time system guarantees that all the kernels working on that batch (and of its internal arrays), even the ones launched by different host threads, are attached to the same CUDA stream in order to enforce the required issuing ordering on the device.

4.4.3 Extending the WindFlow API for GPU-based Operators

Besides an efficient run-time system leveraging GPUs, WindFlow has been designed with the purpose of being user friendly through a new high-level API hiding the complexity of running streaming operators on accelerators. To this end, WindFlow masks completely to the application programmer all configuration aspects of CUDA kernels (e.g., block size, number of blocks) and of host-to-device data transfers (whether they are explicit or implicitly defined through the use of CUDA unified memory).

The API is essentially based on a set of builder classes that can be used to instantiate Map and Filter operators working on the GPU. The following code fragment shows an example of instantiation of a Map operator. As we can observe, the user responsible of providing the business logic code to instantiate the operator, which is in this case a `__device__` lambda respecting a required signature for the Map. Other configuration parameters are the usual ones already used for traditional WindFlow operators, like the parallelism degree (number of operator replicas implemented by separated host threads), a string used to identify the operator for logging purposes, and so forth.

```
Map_GPU map = MapGPU_Builder(__device__ [](input_t &t) -> void {…})
   .withParallelism(2)
   .withName("MyMapGPUOperator")
   .withKeyBy(__host__ __device__ [](const input_t &t) -> size_t {…})
   .build();
mp.add(map);
```

In the fragment, the Map is created with a keyby extractor, i.e., a function (both callable from the host and from the device) used to extract a key attribute (in this example an integer) from an input item. The presence of the key extractor in the builder obliges the user to adopt a stateful signature for the function (a lambda) provided to the builder constructor, whereas a stateless signature must be used in case of a stateless version of the Map. All these checks are done at compile time, using proper static asserts and template metaprogramming tricks to enforce the required constraints and produce helpful messages to the user during the compilation steps.

4.4.4 Preliminary Evaluation

The preliminary evaluation has been done using two benchmark applications from the streaming domain:

1. **SpikeDetection** (SD) is a financial application that finds out the spikes in a stream of sensor readings using a moving-average operator and a filter. It is a logical pipeline of four possibly replicated operators: a source, a stateful map (maintaining a moving window per key

attribute), a filter and a sink. The stateful map and the filter are executed on GPU while the other operators are kept on CPU.

2. **FraudDetection** (FD) applies a Markov model to calculate the probability of a credit card transaction being a fraud. It is a pipeline of three stages: a source, a stateful filter and a sink. The filter is implemented on GPU (with a complex state object keeping the last observations per key and a matrix of probabilities) while the others are on CPU.

The evaluation compares the throughput experienced by running all the operators of the two applications on the four cores of the ARM processor (Jetson Nano), and the one obtained by using (together with the ARM CPU) the integrated Nvidia GPU available on the SoC. The evaluation has the purpose of providing a first assessment of the efficiency of the implemented mechanisms for GPU offloading of hotspot operators. In addition, we compare different variants of the GPU-based version:

1. **Explicit transfers**: it is a variant not exploiting the physical memory sharing of the SoC, so using explicit transfers between host-accessible and GPU-accessible memory buffers.
2. **Unified memory**: it is a variant using the CUDA unified memory, so it avoids unnecessary transfers.
3. **Unified + Prefetching**: it is a variant of the previous one enriched with prefetching CUDA directives, to help the coherency mechanisms of the CUDA driver.

Results are summarized below in Figure 19 for SpikeDetection (top) and FraudDectection (bottom). As we can see, throughput increases using larger batches. Thousands of inputs per batch are necessary to amortize the run-time overheads. The version with prefetching reveled the best alternative in both the applications, while the version with explicit transfers performs worse as expected.

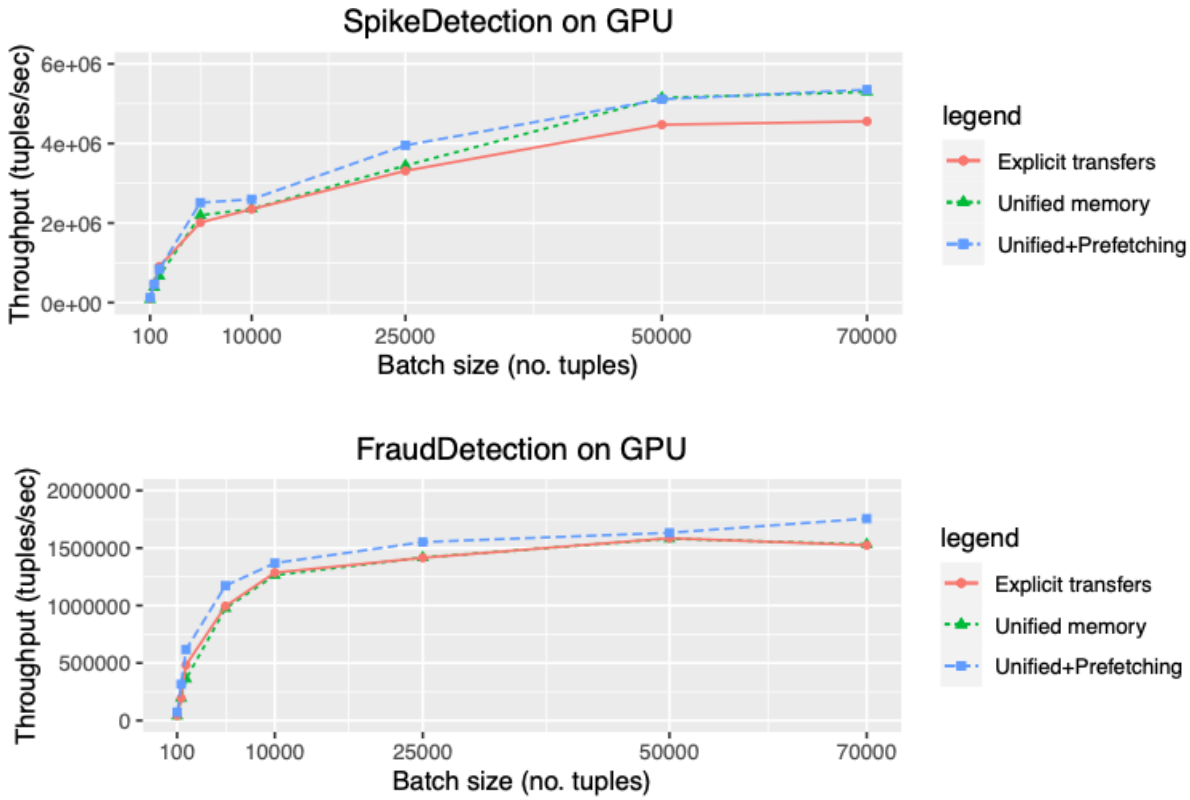


Figure 19: Experimental results on the Jetson Nano using the integrated GPU.

The comparison against the version using only the CPU cores (without offloading tasks to the GPU) is remarkable. For SpikeDetection, we measure a throughput increase of 1.8 times while for FraudDetection it is of 3.12 times. The improvement however strongly depends on the workload characteristics, so these preliminary results will be extended in the next steps of the T2.2 activities in order to improve the evaluation and enhance the tuning of the run-time system.

4.5 V2X Networking Framework

Previous Section 2.3 in this document introduced the Vehicle-to-Everything (V2X) networking framework and related recent literature and results. In this section we discuss how the V2X networking framework is applied to support the HPC2I platform and how it matches TEACHING needs.

V2X adoption allows exploiting edge resources to support the definition of a machine learning approach, organized accordingly with the federated learning paradigm, over a set of federated Cyber Physical Systems (CPSs) targeting wearable sensing systems. We focus on a specific CPS involving a smart-vehicle environment, where the key investigation target is not the advanced driver-assistance systems, but the human perception of the driving style of the autopilot. The top-left corner of **Error! Reference source not found.** shows the core CPS, which we use as a reference, and is made up of data producers:

1. the user-wide-wearable-collection (UW2C).
2. the vehicle, whose data streams include logs exposed from the car *CAN bus*[56] as well as from embedded sensors like LIDARs, dash cameras, obstacle detection units, etc.
3. a data storage, to preserve, locally, the privacy of data
4. a local computing platform (in blue) deputed at training local models and performing local inference to characterize the physiological, emotional, and cognitive (PEC) state of the passengers.

Such computing platform, according to our vision, is expected to be highly flexible and configurable to also support different learning paradigms, i.e., in the case of models trained remotely or in a distributed fashion, the computing platform can just conduct data pre-processing and compression, or to coordinate its work with remote and federated resources. It is also worth specifying that our model does not nail a core CPS within the boundary of a vehicle. In fact, while keeping it as a possibility, core CPS is a more general model whose physical breakdown can involve different entities, such as Edge computing devices, e.g., Road Side Units (RSUs).

The core CPS can take advantage of the computational power and storage on the cloud in order in order to offload some tasks. There will be a balance between

- computations that can be offloaded to the cloud and
- computations that need to return or remain on the edge, for the reasons aforementioned.

[56] Details on the CAN bus can be read at: http://can.bosch.com

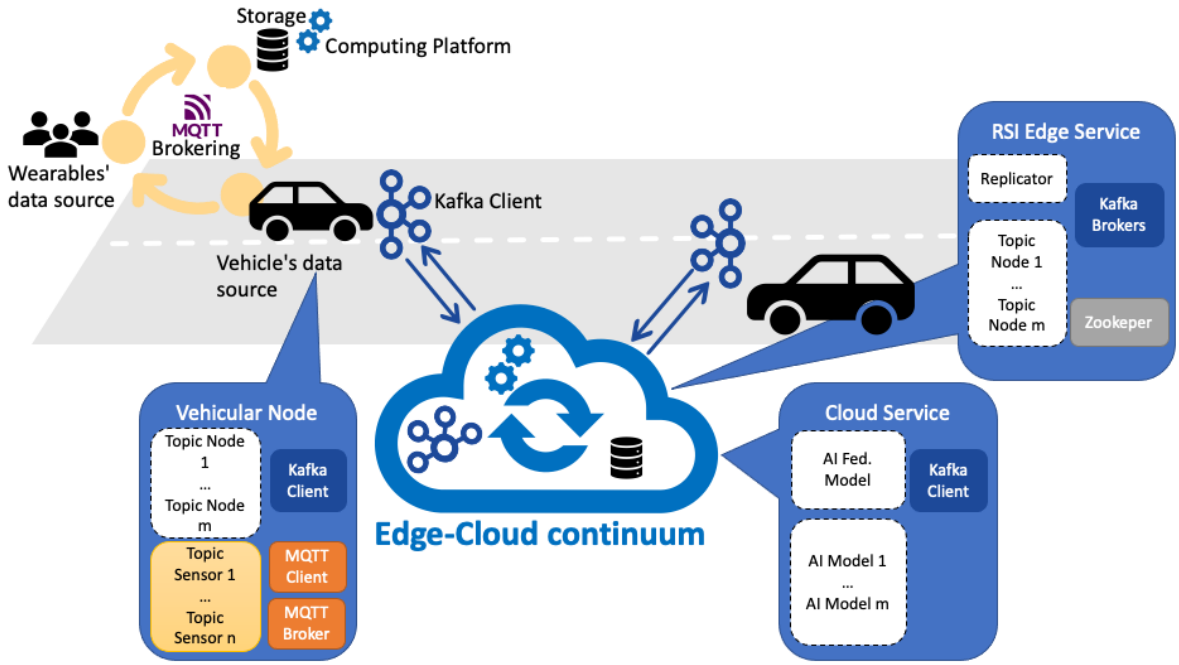


Figure 20: Reference Use Case, Multi-Access Communication and Computing Infrastructure

In addition, different instances of core CPSs as in **Error! Reference source not found.**, can cooperate through the cloud (in blue), with other analogous systems to implement a federated learning approach. This means that different core CPSs share the hyper-parameters of the locally computed PEC models and synthesize a new, global PEC model to be re-distributed to all the CPSs for further refinements.

In these settings, privacy control is implemented by design, as no raw personal data coming from wearable sensors or from the CAN bus are disclosed outside the core CPS. As a matter fact, the complexity derived from generalizing the generalization of the core CPS concept might lead to a very complex management of the interactions of the modules composing it. In fact, being potentially scattered across different physical entities, introduces potential issues, such as: connectivity, timely interactions, etc. To overcome this limitation an interesting conceptual tool, that can be used when modeling and controlling CPS, is that of Digital Twin (DT). A DT is a detailed, usually multidisciplinary virtual model that completely reproduces the behavior of a real physical equipment or system[57], the DT, whose approach is discussed for industrial processes. In our use case, the DT might be associated with a UW2Cs, or to a complex monitored system, like a smart vehicle. The DT may rely on mathematical models or on synthetic models derived through deep neural networks training.

The overall scenario presented in **Error! Reference source not found.** matches the Multi-Access Edge Computing approach we discussed in Section 2.4. As explained there, the TEACHING project needs to deal with the environment characterized by the MEC approach, where ultra-low latency and high bandwidth links as well as real-time access to the radio network can be and must be leveraged by applications. The scenario specifically fits the

[57] Radchenko, G., Alaasam, A., & Tchernykh, A. (2018). Micro-workflows: Kafka and kepler fusion to support digital twins of industrial processes. In *2018 ieee/acm international confer- ence on utility and cloud computing companion (ucc companion)* (pp. 83–88).

automotive use-case within TEACHING, where MEC platforms allow designing infrastructures where data acquisition, data pre-processing and training of local Artificial Intelligence (AI) models can be performed by leveraging the edge resources placed closely to the Road-Side Infrastructure (RSI). The idea of the Edge-Cloud continuum applied in our scenario allows performing out of the vehicle some of the computing operations, which can reduce significantly both the computational load on the vehicle hardware and the response latency of the services for the end-user, optimizing the user experience.

The infrastructure can thus rely on computing entities on board the vehicles and on the RSI. In the former case the main target are lightweight computing operations such as raw-data pre-processing, mapping applications, and navigation programs. Due to the limited computing resources that are typically available on a vehicle, complex computation tasks can be offloaded to the servers of the RSI, or collaboratively shared among different vehicles, if the communication infrastructure support this possibility. Indeed, RSI provides a broader range of services, since edge servers have more computation resources than a typical vehicle. Edge servers can provide computing resources to vehicles in the relative coverage area, as well as handle all the operations that the vehicles offload. When all the edge servers in the coverage range are busy, the RSI can, in turn, offload the tasks either to the Cloud or to intermediate servers towards the core network.

4.5.1 INFRASTRUCTURE DESIGN

This section provides an insight into the proposed network architecture, by introducing the available communication technologies, network, and transport protocols and platforms to implement an event streaming paradigm. Section 4.5.1.1 discusses the underlying technologies, Section 4.5.1.2 connects with the HPC2I architecture. Section 4.5.1.3 discusses the in-vehicle data management in the context of the HPC2I platform. Sections 4.5.1.4 and 4.5.1.5 provide information on how the MQTT and Kafka pub/sub protocols are used within HPC2I and as a bridge toward the AIaaS architecture of WP4.

4.5.1.1 Underlying Communication Technologies

The proposed architecture is based on two principal technologies: MQTT[58] and Apache Kafka[59]. MQTT is a publish-subscribe network protocol whose goal is enabling the exchange of messages between devices, usually over TCP/IP. Its main target are connections with remote locations when the network bandwidth is limited. Message Queue Telemetry Transport (MQTT) choice is driven by the large diffusion of MQTT's APIs for constrained devices and relative applications in the context of the Internet of Things (IoT), as far as most of the wearable devices expose an MQTT client to publish the produced data.

Apache Kafka is an open-source platform for distributed event streaming messaging. While traditional messaging systems such as IBM Websphere MQ[60] provided a very high level of guarantee of message delivery (as each message needs to be acknowledged), such a strict reliability constraint is often overkill for the log collecting process. Conversely, data loss cost is assumed negligible in Kafka, thus reducing the complexity of both APIs and implementation. A common denominator of messaging-based log aggregators is the organization of data streams

[58] MQTT: https://mqtt.org

[59] Kafka: https://kafka.apache.org/

[60] IBM Websphere MQ: https://www.ibm.com/support/knowledgecenter/

in topics. As for MQTT, also in Kafka, a producer can publish messages on a topic. Yet, published messages are managed by brokers.

A relevant difference that distinguishes Kafka from other platforms is in the distribution policy of messages. Most of the messaging systems, like MQTT does, support a *push* policy for messages that the broker has to deliver from producers to consumers. Kafka, instead, applies a *pull* policy in message forwarding: each consumer can try to retrieve data at the maximum rate that it can handle but is never flooded by messages. Another design choice that makes Kafka very scalable and suitable for distributed systems, is that it can exploit multiple broker instances. To simplify the coordination of the instances, Kafka employs Zookeeper[61] in order to

i) detect the addition and the removal of both brokers and consumers,
ii) maintain the relationship among brokers and consumers,
iii) trigger a re-balancing of the workloads when either brokers or consumers are added or removed.

In our architecture, KAFKA components run partly on the mobile node and partly on the far edge. The Client component is used to develop the part of code that runs on the mobile node (Far Edge) and allows its connection with the infrastructure of KAFKA brokers; such brokers are implemented in the Near EDGE infrastructure. Precisely, a Consumer module has been developed to read the data published on the MQTT broker from the various sensor networks used onboard the mobile node or worn by the user. The Consumer, therefore, behaves as a subscriber of MQTT topics. Consistently exploiting the Client component, a Producer component has been developed to publish the messages of the MQTT topics on the KAFKA topics. Here the topics have only one partition that acts as a leader and is replicated a predetermined number of times on many brokers. These replications are the followers to give redundancy to the information. The Consumer module also reads the aggregated/federated model coming from the Cloud or the Near EDGE, or even from the Far Edge on behalf of the AI module, while the Producer module can publish this model in an MQTT topic if it is necessary to have it read by devices on board the vehicle that communicates through MQTT. Interoperability between KAFKA and MQTT is guaranteed through the KAFKA Connect component that, through the Sink and Source modules, can make KAFKA communicate with systems that do not speak KAFKA.

The KAFKA's component configuration is maintained on the Zookeeper platform located on the Near infrastructure or in the EDGE Cloud. Precisely in Zookeeper are maintained:

- The information is used by the KAFKA Controller, which is responsible for maintaining leader-follower relationships across all partitions. If a node for some reason goes down, it is the responsibility of the controller to tell all replicas to act as partition leaders to fulfil the duties of partition leaders on the node that is about to fail.
- Information about KAFKA brokers needed to elect a new Controller if the current one goes down. In Zookeeper it is maintained all information about active topics on KAFKA brokers, the number of partitions for each topic, the position of all replicas, the list of configuration overrides for all topics and which node is the favourite leader.
- The list of all brokers that are running at a given time and are part of the cluster.
- Access control lists (ACL) of all active topics on the brokers.

[61] Zookeeper, http://hadoop.apache.org/zookeeper

Note that the best practice is to exploit the redundancy of the configuration information provided by the Zookeeper repositories to avoid that the fault of an entity implies the fault of the whole infrastructure.

In our infrastructure, to allow the auto-configuration of a new node, the new node has been registers through a portal and download the configuration file, which contains the IP addresses of at least one KAFKA server, where the new node can request all the info such as topics, partitions cluster. The configuration procedure also allows the download of the certificate that the new node will use to authenticate itself with the broker. So, the infrastructure needs of a Certification Authority. KAFKA uses for its client-broker communication the TLS protocol. Once the node is connected, it receives info on the active brokers and topics, then new node and broker exchange their certificates. Once acquired, the respective certificates broker and new node communicate in an encrypted way. Note that the KAFKA ACLs, in our case, is used in such a way to avoid that each node can read the topics where its federated model is saved and cannot go to read the other ones.

4.5.1.2 Medium Access Architecture

In the HPC2I architecture we foster the application of C-V2X technology we discussed in Section 2.3. As C-V2X allows for faster and easier deployment of supporting communication infrastructures by simply upgrading the existing base stations, it preserves the continuity of legacy services as well as offering a path toward pervasive coverage on a global scale. These are the two key elements toward larger and larger adoption of the C-V2X technology.

Within the HPC2I architecture, we will generally refer to a RSU like a generic communication device belonging to the roadside infrastructure include at least the hardware and services of an eNB in Mode 3 and possibly more.

4.5.1.3 In-Vehicle Data Management

The design of the communication scenario is based on the scheme shown in Figure 21. It highlights three main entities:

(i) Mobile Nodes standing for the Far Edge.
(ii) RSI Edge services identified as the Near Edge.
(iii) Core Cloud services.

The mobile node is equipped with an MQTT broker to publish, locally, the data produced by the vehicle's sensors and UW2Cs. Data is published into a set of topics, such as: *private_data*, *shared_data* and *AI_model*. Data destined to in-vehicle processing can be published onto the *private_data* topic; data that can be shared with other vehicles or network services can be published onto the *shared_data* topic. Finally, the model topic is devoted to data concerning the *AI_model* shared among the entities of the infrastructure.

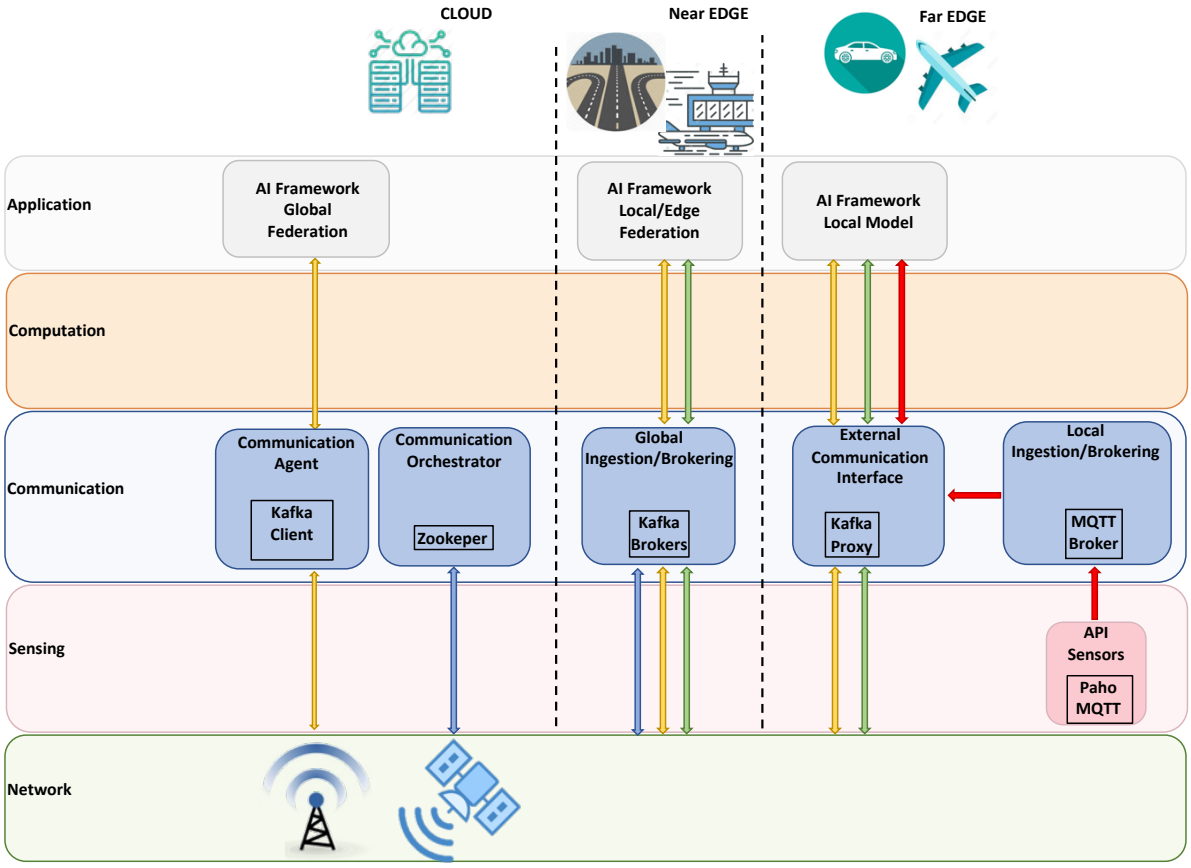


Figure 21: HPCI Networking Reference Architecture.

Data topics are accessed by both local and remote subscribers, the latter will be analyzed when we discuss the Cloud-service entity (Section 4.5.1.5). Local subscribers, on the vehicular node, are devoted to services such as AI algorithms, database management, and a Streaming Client (SC), whose development through the Kafka platform. Note that, the set of local subscribers can vary in number and functionality, but a SC is a needed component of the proposed infrastructure. Indeed, Kafka enables the duplication of the topics on-board the mobile node towards the edge resources, as detailed in Section 4.5.1.5.

4.5.1.4 MQTT Communication paradigm

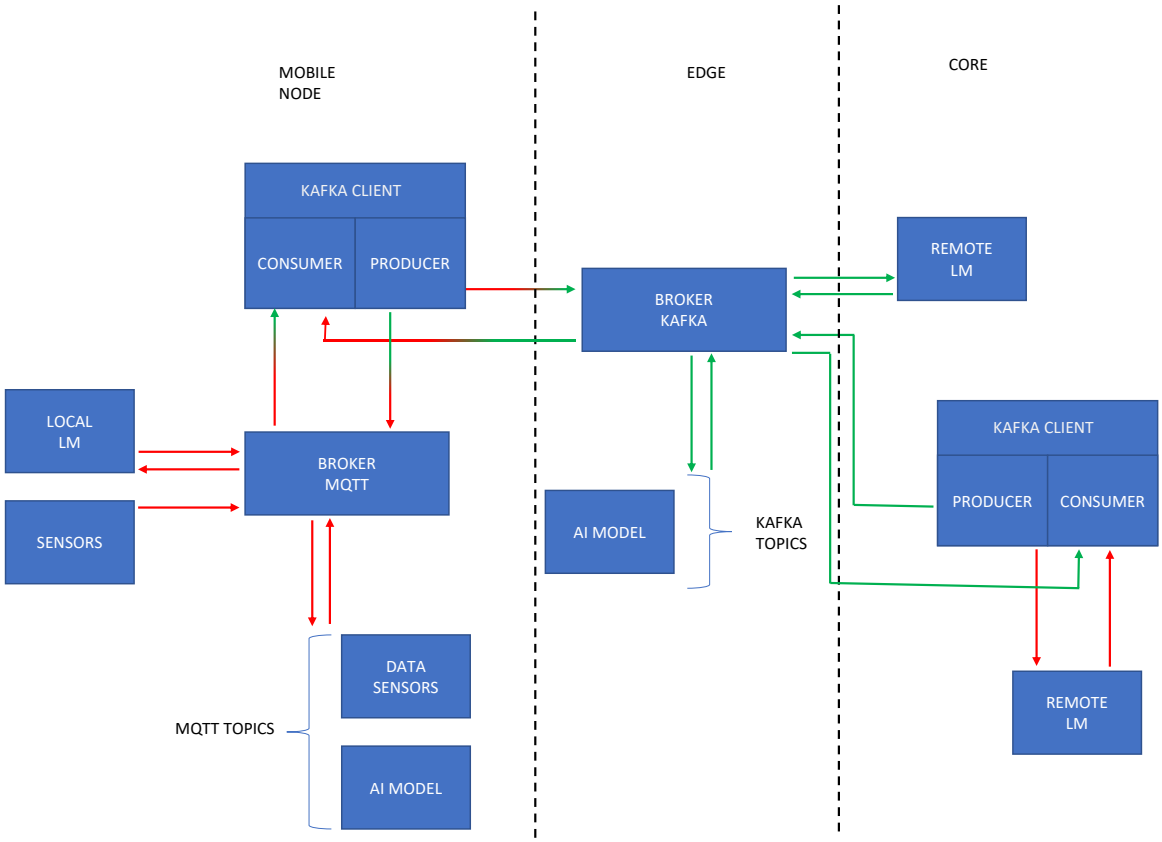


Figure 22: Local/Global Ingestion/Brokering Communication Schema

Figure 22 shows that the learning modules available onboard the mobile node to communicate through the infrastructure can use the MQTT protocol.

The MQTT broker will mostly act as a proxy for the Kafka protocol, on the ground that MQTT is a more lightweight protocol and broker, which can cross the network but is less optimized for the task. MQTT offers some degree of QoS configurability and support for reliability, as the example in Section 6.1 will show. On the other hand, the Kafka protocol and the brokers implementing it provide mechanisms to deal with more complex replication, reliability and networking latency optimization problems, hence they are ideal for the long-range networking within the HPC2I platform.

The situation will be reversed in deliverable D4.2, as there the MQTT plays the central role in the local system (the AIaaS instance possibly within a given CPS) while Kafka based networking is mostly an interface toward the "external world".

4.5.1.5 Kafka Communication Paradigm

Kafka is a distributed publish-subscribe messaging system that maintains feeds of messages in partitioned and replicated topics. In the simplest way there are three players in the Kafka ecosystem: producers, topics (run by brokers) and consumers as shown in Figure 23. Producers produce messages to a topic of their choice. It is possible to attach a key to each message, in which case the producer guarantees that all messages with the same key will arrive to the same partition. Topics are logs that receive data from the producers and store them across their partitions. Producers always write new messages at the end of the log. In our example we can

make abstraction of the partitions since we're working locally. Consumers read the messages of a set of partitions of a topic of their choice at their own pace. If the consumer is part of a consumer group, i.e., a group of consumers subscribed to the same topic, they can commit their offset. This can be important if you want to consume a topic in parallel with different consumers.

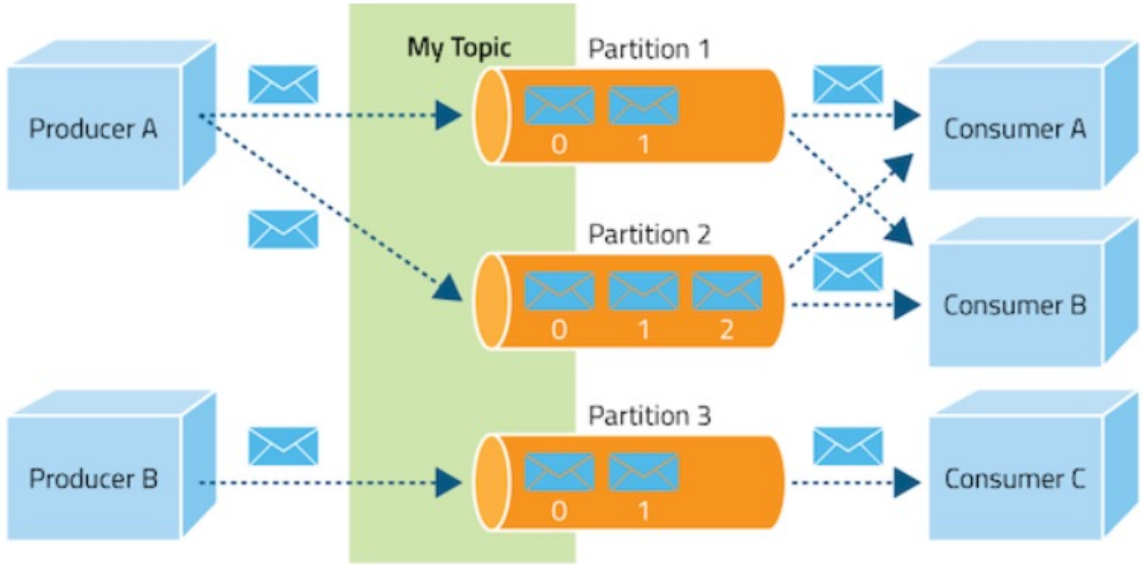


Figure 24: Kafka data producers, data consumers, and partitions. Figure source https://www.cloudera.com/documentation/kafka/1-2-x/topics/kafka.html

The offset is the position in the log where the consumer last consumed or red a message. The consumer can then commit this offset to make the reading 'official'. Offset committing can be done automatically in the background or explicitly.

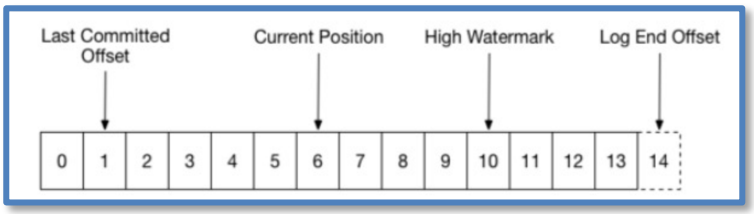


Figure 23: Offset of the consumed messages in Kafka. Source: https://www.confluent.io/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/

The implementation of the TEACHING HPC2I interface with the Kafka protocol was developed in Java with the aim of being used both in the HPC2I general settings, as well as in the more specific case of the AIaaS architecture developed in WP4 (see deliverables D4.1 and D4.2). An example of its use is shown in this document in Section 6.2. The code and documentation have been made available to the partners on the TEACHING GitLab, and was exploited so far by WP2 and WP4 and tested in the project integration meetings.

4.5.2 V2X Data Streaming

The architecture we propose assumes that edge servers are co-located with RSUs. The entities of the RSI resulting from such a combination provide to Vehicular Nodes both network access and computing capabilities. The Kafka streaming client was introduced in Section 4.5.1 and is also shown in Section 6.2. The Kafka streaming client on the Vehicular Node can work as publisher toward the Kafka broker of the road infrastructure, thus replicating the data associated with the subscribed MQTT topics on the Edge. A topic replicator is installed in the RSU, which allows replicating broker's managed data (i.e., topics) over a set of federated brokers, allowing to increase the resilience of the RSI toward the possible faults of the deployed brokers.

Using such a streaming paradigm, a Kafka collection of brokers acts as an intermediary between the Cloud and the Vehicular Nodes, as it can store the messages originated from the end-users toward the Cloud and, conversely, it can deliver the responses coming from the Cloud toward the Vehicular Nodes. Placing the Kafka cluster on the RSI, brings two main advantages:

- the latency between edge nodes and vehicular nodes is reduced,
- resilience is increased (w.r.t. ordinary cloud services) if the Cloud and Vehicular Nodes are decoupled through a publish-subscribe paradigm, in our design implemented by the collection of Kafka brokers.

Vehicular Nodes, moving on a geographical area, where such a communication infrastructure is deployed, will always be able to interact with the redundant brokers through the RSUs. Zookeeper is exploited and configured in order to allow this, thus maintaining configurations, naming, synchronization, and grouping of the services for all the brokers installed on the RSI. Zookeeper keeps a copy of the state of the entire distributed system, by using local log files, to keep the persistence of this information over time. All these functionalities allow propagating the updating of both data and configurations of brokers on the RSI.

4.5.3 Supporting Distributed and Federated Learning

The Cloud Service is the most remote computing entity of the proposed infrastructure, from the perspective of the vehicular node, i,e., the data producer. This entity is devoted to achieving the federation of the AI models trained on the Edge. Generally speaking, here the term Cloud is not necessarily to be identified with public Cloud services (e.g., AWS, Google Cloud, etc.), but can as well mean private Cloud (e.g., a data-center provided by automotive vendors).

The algorithm that runs on Cloud performs the federation of the AI models trained on the Edge. As such, it is the remote subscriber of the topic *AI_model* in Kafka. Meanwhile, the Kafka client operating in the cloud service is a publisher, behaving according to a "push" semantics, on the topic *AI_model* to deliver the federate AI model, back to edge servers on the RSI through the Kafka brokers. Kafka clients, running on the vehicular nodes, adopt instead a "pull" semantics in retrieving the updated federated model from the RSI. This allows the AI agents on the vehicles for updating their local models. In order to preserve privacy allowing each vehicular node to post and subscribe only its own topics, A Kafka subscriber can gather on a topic only after receiving proper authorization. For example, only the cloud service is allowed to publish the AI federated model on the *AI_model* topic towards all the Edge subscriber nodes. We achieve this goal by imposing that the communications between publishers, subscribers, and brokers are encrypted using Transport Layer Security (TLS), with certificates issued by a private Certification Authority (CA). These certificates are uploaded on target nodes, i.e., on Edge nodes and the relative Kafka clients on the vehicular nodes. These credentials, produced for each node through the certificates, can be used to achieve an Access Control List for Kafka

brokers. In this way we can configure the network with a small set of topics onto which computing services can perform operations without privacy issues.

4.5.4 Evaluation of the Intermittent Connectivity

The performance evaluation of the proposed infrastructure is based on the scenarios shown in Figure 26 and Figure 27. Each figure refers to a different scenario of interest. Figure 26 applies when the adjacent cells of two different RSUs overlap. Conversely, Figure Figure 27 does apply when the aforementioned cells do not overlap. Table 4 reports the parameters considered during the performance evaluation. Parameter selection has been conducted by relying on the state-of-the-art results[62],[63].

Table 4: Parameters of the Simulated Scenario

Parm.	Val.
R (m)	1000
Δh (m)	30
$\overline{FC}$ (m)	$\{0, 350\}$
$\overline{O_A B}$ (m)	400
T_{HND} (ms)	14.5
C (b/s)	$6 \cdot 10^6$
P_{Tx} (dBm)	23
f_{car} (GHz)	5.86
MCS	5
v (Km/h)	100
δ_{vhc} (Vehi./Km)	18
λ (msg/sec)	100

The performance evaluation focuses on a freeway mobility scenario, such as a highway. We assume that vehicles move along the trajectory with constant velocity v, and that the linear density of vehicles per meter of the trajectory is δ_{vhc}. In this mobility scenario, we also assume that the coverage range of the cell is R, the height difference of the antennas between the vehicle and the cell is Δh. The cells can overlap each other for about 1/3 of their radius and the minimum distance between antenna and end-user is $\overline{FC}$. The link capacity is C, the time required for the handover is T_{HND}, the transmission power is P_{Tx}, the frequency carrier is f_{car}, and the adopted modulation and coding scheme is *MCS*. Finally, we assume that the mobile users can generate data messages at a rate λ. We also observed that Kafka generates messages for the transmitted data with a fixed size *msg*, and a new message is sent when the delivery of the previous one has been confirmed by an acknowledgment message *ack*. The analysis is conducted on traffic traces generated by Kafka, which have been dumped through Wireshark

[62] Toghi, B., Saifuddin, M., Mahjoub, H. N., Mughal, M. O., Fallah, Y. P., Rao, J., & Das, S. (2018). Multiple access in cellular v2x: Performance analysis in highly congested vehicular networks. In *2018 ieee vehicular networking conference (vnc)* (p. 1-8). doi: 10.1109/VNC .2018.8628416

[63] Singhal, D., Kunapareddy, M., Chetlapalli, V., Vinosh Babu James, & Akhtar, N. (2011). Lte-advanced: Handover interruption time analysis for imt-a evaluation. In *2011 international conference on signal processing, communication, computing and networking technologies* (p. 81-85). doi: 10.1109/ICSCCN.2011.6024519

packet sniffer. The analysis highlights that the time required to: *(i)* establish an encrypted TLS connection between the client and the broker, *(ii)* and receive the first message *Kafka Produce Request* is in the range $T_{con} \in [0.094, 0.322]$ *s*.

The traffic exchanged between the client and the broker was TLS encrypted, so we used the Java Agent Library `jSSLKeyLog` to decrypt the traces with Wireshark. `jSSLKeyLog` allows logging the TLS session keys on a file, which can be loaded on Wireshark during the decoding of packets associated with Kafka messages.

We use a Java routine to generate data messages for the topics using a fixed data rate λ: these messages are published in the topic stored on the vehicle. To this end, we used the Mosquitto broker [64] implementation of MQTT.

The Kafka client is a Java routine that forwards messages. It takes messages from the topics on the vehicle and sends those to the Kafka brokers on the Edge RSI. A ZooKeeper service has been configured to support the Kafka brokers, aiming at managing the synchronization of the brokers' replicas. Kafka broker infrastructure and the routines on the vehicle service have been run on two separate machines, based on Linux Ubuntu 18.04. Communication between Kafka brokers, between a Kafka broker and the Vehicular Kafka client, and between a Kafka broker and ZooKeeper are encrypted via certificates generated by a private Certificate Authority. The simulation of the mobility of the vehicle and of the communication link between vehicle and Kafka brokers have been implemented with python-based routines. Table 5 shows the results of the performance evaluation, that we discuss in the following.

Table 5: Summary of the Performance Evaluation

Parm.	Overlapped use case	Non-Overlapped use case
T_{ON} (s)	59.17	65.7
T_{OFF} (s)	0.0145	6.2
Q_{TX} (pck)	0.34	630
μ (ms)	12.3	25.5
msg/ack (B)	343/161	343/161
T_{CON} (ms)	[0.094,0.322]	[0.094,0.322]

For the scenario in Figure 26 we have measured the transmitting time interval T_{ON} that is the length $\overline{DA}$ divided by the velocity *v*.

[64] Mosquitto: https://mosquitto.org

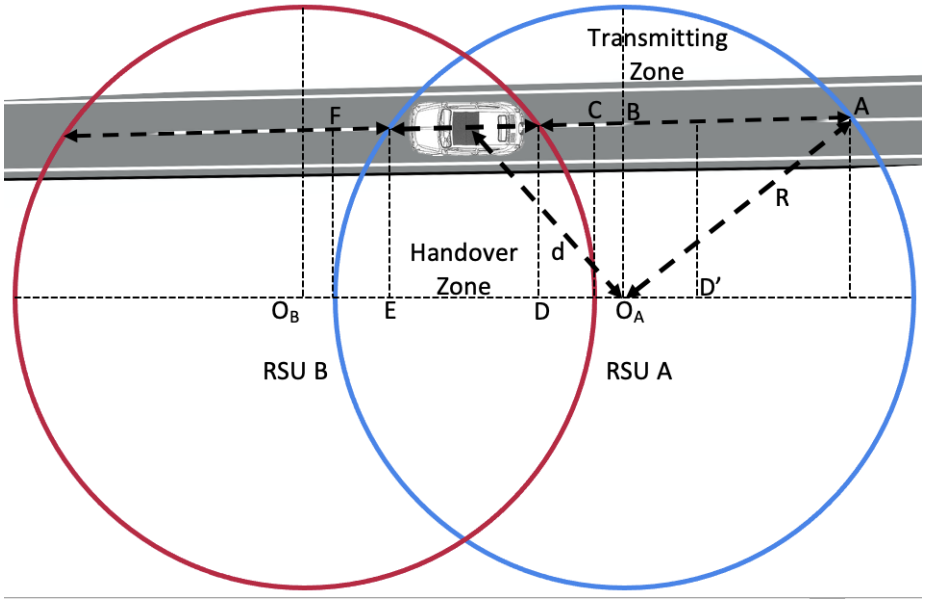


Figure 25: Overlapped use case

Note that, at the time T_{ON}, we need to subtract the T_{HND} (handover time between the previous cell and the current one) and, in the case of the first encountered cell, also the time T_{con}, which is required by Kafka to establish a client-broker connection. When the mobile user is within the handover zone the interval time without transmitting data is $T_{OFF} = \min(T_{HND}, ED/v)$. For the parameters reported in Table 4 we obtain a value $T_{ON} = 59.17\ s$ and a $T_{OFF} = T_{HND}$, precisely, for the simulation of the handover we use T_{OFF} as the parameter of the exponential distribution. We observed that the queue of the transmitter has been almost empty throughout the simulation time, which results in an average queue length Q_{Tx} of 0.34 packets and leads to a packet pacing μ of 12.3 ms on average. Given the generation rate of λ=100, corresponding to a packet pacing of 10 ms, this means that the inter-time of the information at the receiver increases by 23%.

Regarding the scenario in Figure 27, we have obtained a transmitting inter-time T_{ON} as the length $\overline{E'A'}$ divided by the velocity v. Note that, we need again to subtract the time T_{con} from the time T_{ON}, to account for the Kafka connection time between the client and the broker. Indeed, in this case the connection is lost every time the vehicle leaves the cell. The time $\overline{A'A} / v$ is the time to cover the distance on the trajectory between the vehicle and the antenna, which is greater than R. In this scenario $T_{OFF} = \overline{F'E'} / v$. For the parameters in Table 4 we obtain the values $T_{ON} = 65.7\ s$ and a $T_{OFF} = 6.2\ s$. We observed that the queue of the transmitter has an average length of 630 packets, and the packet pacing is 25.5 ms on average. Again, for

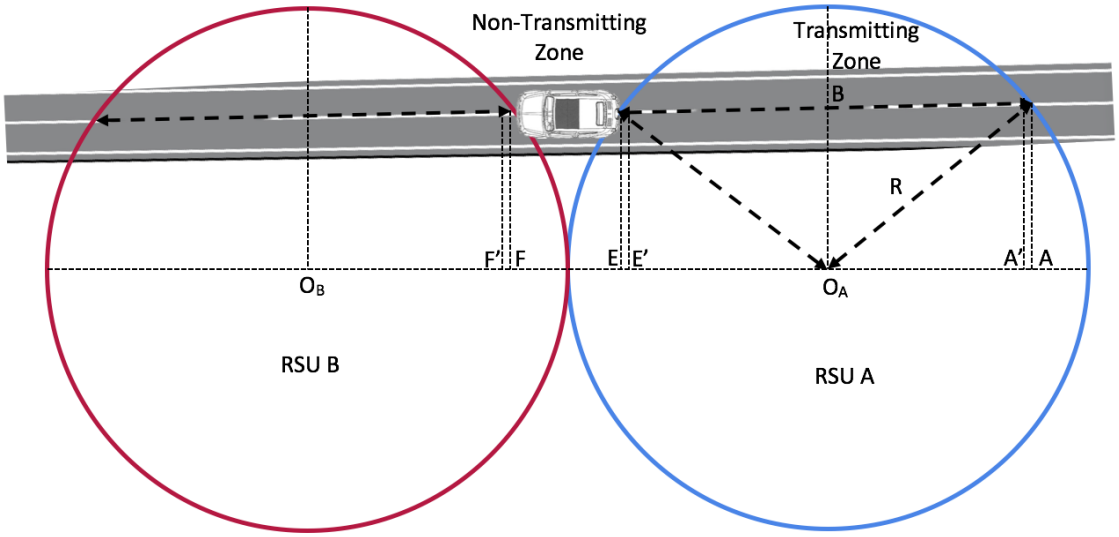


Figure 26: Non-Overlapped use case

a given message rate of λ=100, the inter-arrival time of Kafka messages at the receiver is more than doubled (1.5 times).

4.6 Application deployment and Resource management

A properly designed management process is needed to put together and handle the hardware and software resources characterizing HPC2I so that their placement can be flexible and depending on the actual needs and indications of application providers to be put strategically near the data source or where certain kinds of resources are more available.

As described in the grant agreement, task T2.6 is in charge of providing *"solutions and mechanisms enabling the seamless exploitation of edge and cloud resources, for the provisioning of the computing and storage capacity needed for the filtering, processing and delivery of the data supporting the execution of AI solutions targeting human-centric CPSoS"*

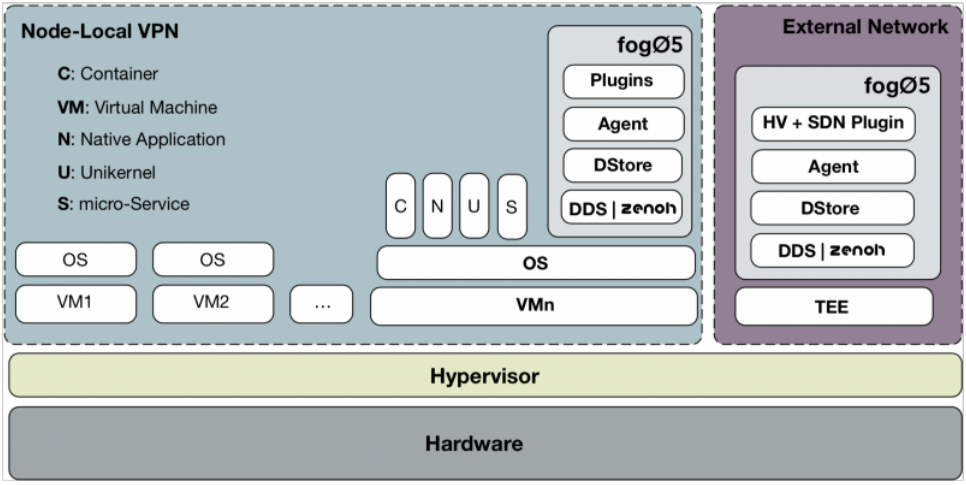
As such, to T2.6 is demanded the management of resource allocation and the workload execution of TEACHING based applications. Coherently with the best practices concerning the provisioning of resources at the edge and in cloud, HPC2I approach is based on resource virtualization and more specifically on microservices running on software containers.

Once virtualized, resources are managed via a Virtual Infrastructure Manager (VIM). Several different VIM implementations with different focuses exist as an example, OpenStack and VMware vCloud Director to manage Cloud Infrastructure, but also OpenShift or Kubernetes (including its lightweight versions) to manage containers.

Beside traditional cloud frameworks, to manage resources hosted in datacenters, there are also solutions with their own VIM developed for different environments, where resources can be limited and have serious constraints from the hardware viewpoint like in IoT and far edge devices. We reviewed the state of the art, and we found several projects working on VIMs that could address the identified issues.

Therefore, in this first part of the activity of Task 2.6 has been screened in more detail some of the most promising solutions to choose the one that better matches our requisites. The evaluated solutions are Eclipse Fog05, LF Eve, Rancher K3S and MicroK8S.

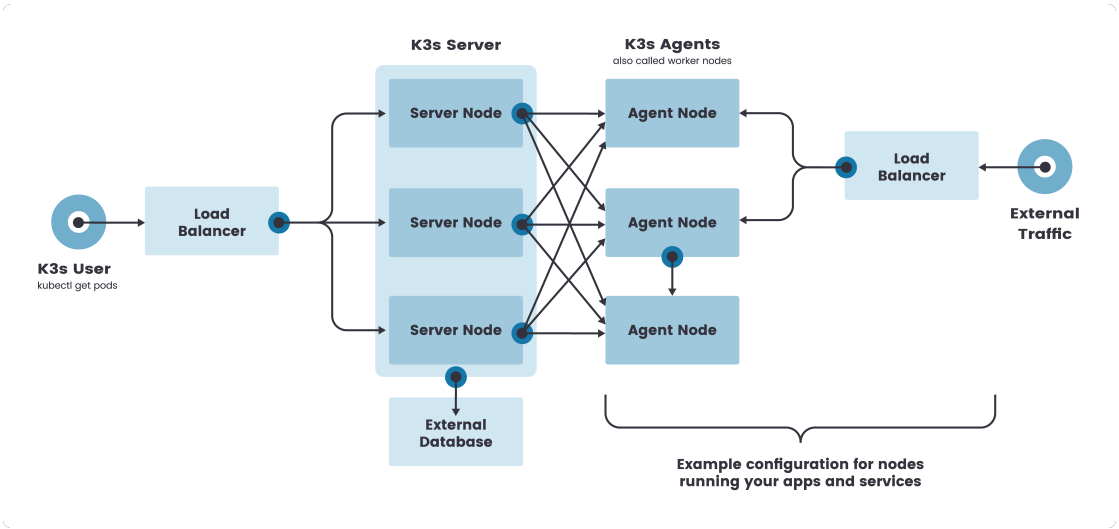
Fog05[65] is sponsored by the Eclipse Foundation. It is characterized by a very simple architecture and revolves around a communication bus based on publish subscribe, focused on keeping a low-overhead profile even using unreliable networks.



LF Eve[66] is hosted by the Linux Foundation and focuses on managing IoT resources, developing a secure-by-design system to run containers that can be managed remotely.

MicroK8S[67] is a Kubernetes distribution for IoT hosted and developed by Canonical.

Rancher K3S[68] is a Kubernetes distribution for IoT.



[65] https://projects.eclipse.org/proposals/eclipse-fog05

[66] https://www.lfedge.org/projects/eve/

67 https://microk8s.io/

68 https://k3s.io/

Our objective is to choose one of the solutions listed above as a baseline for the HPC2I VIM component. This VIM should be easy to be extended, able to work well both on constrained hardware such as Raspberry PI and on-premises resources, ensure reasonable security levels, and be able to support different kind of applications and workloads.

The process undertaken has been inspired by the approach that has been adopted by the EU ACCORDION project[69], coordinated by CNR, having its core activity on the definition of a platform for managing a Cloud/Edge computing continuum.

Therefore, the baseline VIM for HPC2I must satisfy a set of requirements in different aspects, to enable further development. It should be possible to extend it and should have an active community working on its evolution. It should be able to manage the kind of limited hardware usually available on the edge as well as standard data center resources. To guide our comparison, we identified the following list of detailed requirements:

- VIM should be able to manage non-homogeneous resources, ranging from quite powerful ones to those highly constrained in terms of computing power, memory, and storage.
- VIM must be able to run on small devices such as Raspberry PI, so it should support both ARM as well as more traditional CPU-architecture (e.g., x86).
- VIM should use the least possible number of resources for cluster maintenance, as in the TEACHING CPSoS, depending on the use case, resources could be limited in capacity and number.
- A security support should be available to enforce confidentiality in components' communication, and to ease authentication and access control.
- VIM must support to run Docker containers, which are the mainstream technology for containerization. However, if possible, the VIM should also support the execution of different types of workflows, such as Unikernels for high start-up performance and maximum efficiency in resources' utilization. It would be a plus to be able to run virtual machines, giving developers the possibility to have an environment that manages both containers and virtual machines can simplify the migration from legacy applications.
- To simplify adoption and diffusion of HPC2I and TEACHING, it's important that our VIM is accessible using standard or industry-recognized interfaces.
- The use of open-source components is also important, as it enables developers to modify and extend functionality independently, without the need to request permission from software owners or pay license fees.

After having identified candidates in the initial screening, to tackle the selection we adopted the following methodology.

- The first step was to define a set of criteria, listed below, to be evaluated for each candidate.
- Then the analysis of each solution followed a sequence of three phases, each phase incrementally increasing our understanding. First, each project's online documentation was accessed to collect all information available, then detailed software documentation looking at the characteristics of the solution and seeking information on the current activity of the project, its developers' community as well as its diffusion. After the documentation analysis, we experimented by installing the software in our lab, to better

[69] https://www.accordion-project.eu/

understand capabilities, compatibility, strengths, and weaknesses of each candidate solution.

For each project, we considered if the available features were matching our requirements. Then we extracted the knowledge of each solution by studying its documentation. Some projects are still work in progress and it was necessary to understand their real state of work by filtering all the incomplete or misleading information found in websites, source code and presentations. In most cases, it wasn't straightforward neither to understand if some important feature was present or missing, nor to obtain a roadmap of upcoming features. In some cases, it has been possible to install the solution locally and verify its functionality by directly testing its features.

All in all, to conduct the analysis phases, we collected information following a set of criteria defined in the previous section:

- Documentation: document availability and completeness, from both a developer and administrator standpoint, is important to simplify the solution adoption in TEACHING and let all partners get acquainted with it.
- Installability: the complexity of installation on supported systems is a hint about the product's production readiness; moreover, a simple installation is a plus in itself for its smaller operations costs.
- Licensing: type of open-source license, used to understand if is appropriate and matches the requisite of free reuse and modification of source code, like Apache 2.
- Community: understanding if there is an active community working on the product and supporting it, gives assurances about the software's future evolution and its present support level.
- Portability: is VIM usable on different platforms? In HPC2I, it's important that the selected VIM supports both ARM as well as AMD/x86 architectures.
- Supportability: linked with Community, indicates if support for the software is promptly available.
- Architecture: the simpler the solution architecture, the simpler it will be to maintain it. We looked at the deployment architecture.
- Security Features: this criterion is linked to the need of securing the ACCORDION solution from any unauthorized and malicious access.
- Virtualization Supported: what kind of hypervisors are supported.
- Compatibility: criterion that evaluates how easy it is to integrate this solution with mainstream technology.
- Extensibility: if and how the solution supports extensions.
- Hardware Requirements: how many resources should be dedicated to run VIM processes and are thus stolen from users' workloads

As a result of the selection work, there is a picture of the current solutions for virtualization and clusterization in environments characterized by hardware constrained resources.

The selection process let us to select K3S as VIM baseline. K3S shows good results accordingly with all criteria, especially for what concerns lightweight resource consumption and for documentation completeness for both developers and administrators.

K3S inherits Kubernetes REST API: a de facto standard. This facilitates the interaction for any third-party application. Moreover, it is worth to notice that the K3s project recently joined CNCF (Cloud Native Computing Foundation) increasing its chances to be kept available as an open-source solution.

Another interesting feature is its Software conformance obtained by Rancher, meaning that K3S will keep API compatibility and can be a Kubernetes drop-in.

K3S natively can run only Containers, not virtual machines or Unikernels, so it was necessary to find an extension to support such other types of workloads. KubeVirt[70] matches all the requisites of virtual machine management, has a big community around it, it is also hosted by CFCN and is being used as the base for the Red Hat OpenShift Virtualization product, therefore KubeVirt is our selection as the K3S extension to support Virtual Machines.

4.7 Hardware Behavioural Measurement Tools for CPSoS

In Deliverable D2.1 Section 5.4, we introduced **METrICS**, a Measurement Environment for Multi-Core Time Critical Systems, a framework depicted in Figure 27 proposing an accurate runtime and resource usage measurement with a negligible impact on timing behavior. In this section we present the design and some implementation details of the METrICS software as well as the evolutions and changes that were made to match the needs of WP2 activities and of the use of METrICS in the context of the HPC2I architecture. We shall underline that additional porting effort and customization of METrICS was needed to allow its use within the project use cases. In some of the following subsections we will provide a shorter account and explicitly refer to the content of D5.2 deliverable wherever appropriate.

4.7.1 METrICS architecture

We summarize the design of METrICS, which was already described in Deliverable D2.1, Section 5.4.2. On the software side, the **METrICS library** allows us to insert measurement probes in the safety critical software. The **syscall instrumentation layer** provides a way to automatically instrument each OS system calls. The **collector process:**

- defines a shared memory space to collect measurements
- configures specific measurement scenarios
- transfers the collected profiling information to an external Linux host.

On the hardware side, the **hardware monitor kernel driver** provides the supervisor-level privilege necessary to access to hardware performance monitor counters (PMC), allowing us to count hardware events, including the accesses to shared hardware resource.

[70] https://kubevirt.io/

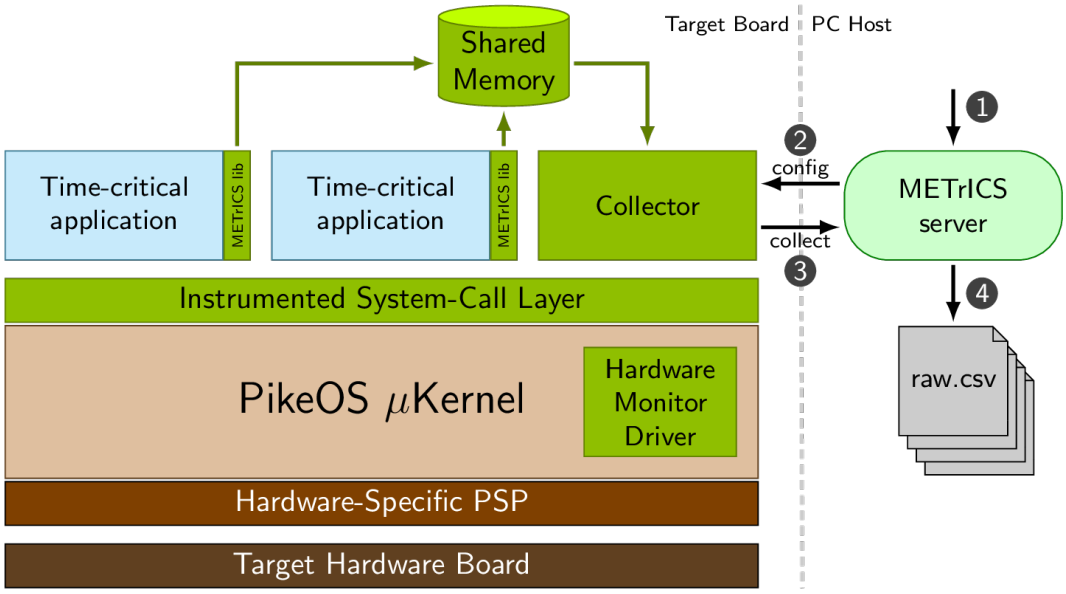


Figure 27: METrICS measurement environment

4.7.2 AI-induced modifications to the trace collection

Before the TEACHING project, METrICS had only been used for hardware and software characterization using statistical analysis techniques. During the TEACHING project, we plan to fully couple METrICS and AI systems to learn the nominal behaviour of applications with regards to the hardware usage, allowing us to detect safety or security issues as deviation from this nominal behaviour, replacing offline statistical analysis with online inference.

As we show in Figure 27, when METrICS was used for a statistical analysis purpose the collection of the traces aggregated by the collector process was performed at the very end, when the execution of the critical application was over. This way, we minimized the timing interference of the collector process toward the critical applications.

Within the TEACHING project, the collected traces are used for two separate purposes:

1. by the machine learning algorithms to build out the neural network and its associated weights from the traces used as input data. Such a neural network would be able to characterize the expected application behaviour on the hardware.
2. by the inference system, on the fly, to detect deviation from the expected behaviour that indicates possible safety issues or security threats.

If the first learning part can still be performed once all the trace data has been collected, as we were doing for statistical analysis, the online inference requires collected data to be passed to the neural network during the autonomous operation of the system, as the data becomes available.

Also, METrICS implies the need to add some probes inside the critical application software with a **metrics_probe_begin** and a **metrics_probe_end** function calls, that are provided by the METrICS library. These probe pairs are added around the software tasks (typically periodic tasks in a real-time system) for them to be characterized in terms of timing and hardware resource usage. Basically, a probe only accesses the pre-set performance monitor counters (PMC) that accumulate how many times a specified hardware event (such as data cache miss)

occurred since the boot of the board. To compute the actual number of cache misses that occurred during a specific task, it is required to pair begin and end probes (most of the time performing a subtraction) to obtain the desired value.

Our pairing algorithm used a specific mask to specify which information should be used for pairing, including if that same time partition was required, the same process id, the same thread id, the same layer (applicative, driver, OS), the same probe identifier the same core number, the same applicative MAF, etc. Consequently, the pairing algorithm was quite complex and time consuming, requiring some data mining in the final generated trace.

For the inference part, TEACHING requires us to perform this pairing online, with minimal timing requirements. Therefore, we defined a TEACHING specific new set of probes that are automatically paired. Such probes are not as much flexible as the previous ones, but do not require any datamining to perform the pairing for probes at the monitored task level.

4.7.3 Modifications to the collection format

Beyond changing the probes, performing the pairing online also had a consequence on the collection format. Such a collection has already been presented in the data management plan (Deliverable D6.2). Performing an offline pairing at postprocessing time allowed to complement the trace with extra information as well as have the pairing algorithm adapt to the type of data: For the timestamp for example, only keeping the difference of the end minus the start time stamp would have only allowed us to obtain the task duration, while losing the starting time that is a critical information when monitoring hard real-time behavior. For some METrICS, doing the subtraction makes no sense as they are not counting some events per se. This is the case for instance with temperature, input / output communication size, and so on...

Performing this operation online requires a little additional embedded computation power, but also some additional storage place in the collection if we need to keep two information instead of one (like starting time and duration for the timestamp aspect). Consequently, a modification to the collection was necessary.

METrICS has already been designed to support multiple record types to fit in the collection. It was originally designed to make the collection record as small as possible to minimize the associated cache accesses, and therefore the timing interference generated by the monitoring system itself. Several record sizes were provided with increasing monitoring payloads, fitting either half a typical cache line size, a full cache line size or two cache lines. Figure 28 shows some example of record sizes.

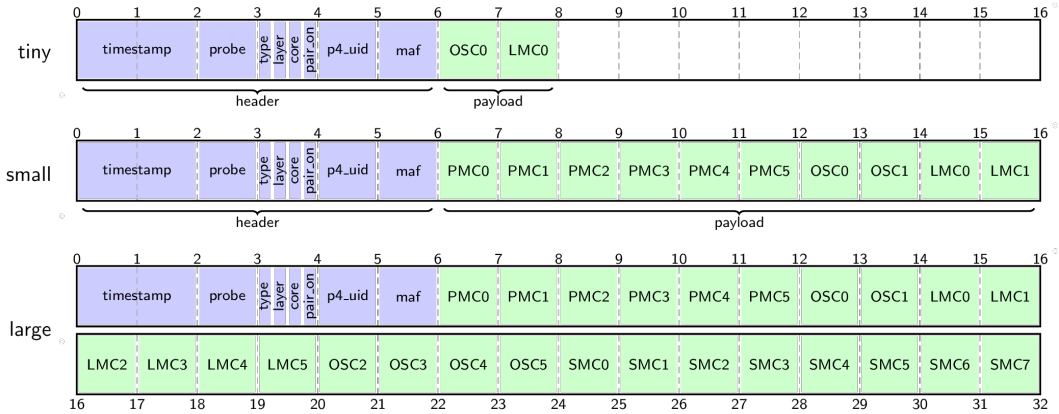


Figure 28: Possible METrICS collection records

The collection records are composed of a common header, as well as some payload with different possible size:

- The tiny record is only filling half a cache line, but it only collects timing information and no hardware behavioral information with the performance monitor counters.
- The small collection is filling a full cache line, it collects the information of 6 PMCs in addition to timing information. Two user-level registers (LMCs) are also provided to allow to add some application specific information at the probe level, such as the iteration count in the software, or some data size information.
- The large record fits two cache lines, it provides some additional LMCs, as well as OS-level registers (OSCs) for OS-related metrics, e.g., counting the number of context switches.

Adding the necessary data to perform the pairing was therefore just about adding a new record type for the TEACHING data. We opted for reducing the number of LMCs and OSCs to add the necessary data, while keeping the record size at the level of a single or a double cache line size.

4.7.4 Hardware behavioral trace format

The traces of hardware events collected by METrICS are aimed to be used as learning material by the machine learning algorithms to figure out the expected normal and nominal behavior of the critical application on the hardware. This first learning phase is performed offline running the application many times in a safe and secure environment.

Once the nominal behavior of the application has been learned in the form of a neural network that could be embedded on the target platform, it will be used during the operation of the autonomous system to check if some deviation could be observed from the expected behavior. Such deviations could be induced by some safety issue, such as the hardware wear-out or by cyber-security attacks that will change the values collected by the PMCs. Spectre and Meltdown attacks for instance will greatly increase speculative branches or speculative memory accesses.

As we can only collect information about 6 hardware events at a time (due to the number of available PMCs), our experimental campaign is split into several experiments that collect information about a particular set of events (See Section 4.2.3 of D5.2 for more details).

For each experiment, traces of hardware events are collected as succession of collection records as presented above in Figure 28, and aggregated as a set of three CSV (comma separated value) files also appearing in Figure 29:

- **hw_event.csv**: This hardware-dependent files list out the hardware events that could be collected by PMCs on the target platform. In the context of the TEACHING project, it is tied to the iMx8 and the Ultrascale+ target platforms provided by I&M. More precisely it is tied to the core type and the associated instruction set architecture (ISA). All the cores on the target platform providing PMC information are Arm-v8 cores that are either Cortex A53 or Cortex A72. They both share the same nomenclature for the available hardware events, even though some hardware events are specific to only on type of core (less speculative-related event son the Cortex A53 that is in-order, as an opposition to the out-of-order Cortex A72 for instance).
  Therefore, this file remains the same for all the experiments performed during the experimentation campaign.
- **run.csv**: This file provides some information about what the particular experiment is with regards to the whole experimental campaign, indicating the particular experiment setup. For instance, it provides information about which hardware events were collected during this particular experiment, as well as which were the contender applications running alongside the critical application, what were this application input data, and so on...
- **raw.csv**: This file corresponds to the trace being collected during one experiment. Each row in the file corresponds to a particular probe pair being added around each task of the critical software to collect the timing and the hardware behavior information of such a task.
  Specifics about the columns semantic has already been presented in the data management plan (Deliverable D6.2). They correspond to a set of information about the probe such as the timestamp when the task started and the duration of the task, as well as a set of collected hardware information (PMC# columns) as detailed in Section 4.7.3.

Figure 29 illustrates the interaction between these CSV files to obtain meaningful information.

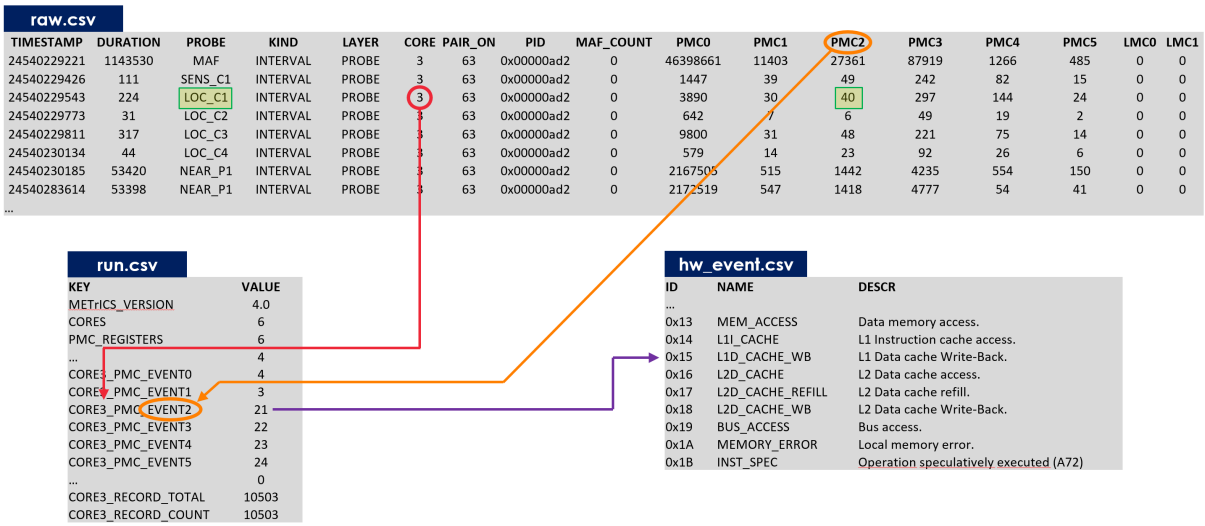


Figure 29: METrICS trace data

For instance, *raw.csv* indicates that for the first iteration of the LOC_C1 task on the fourth core (core number 3), we collected that the third PMC (PMC2) as counted 40 hardware events. The *run.csv* file informs us that such an event for this core in related to event number 21 (or 0x15 in hexadecimal). The hw_event.csv file indicates that this is about counting the number of write-backs from the L1 data cache to the main memory.

4.7.5 Temperature and Power measurement

Within the TEACHING project, to complement the performance information provided by METrICS, we developed **THErM (Temperature Health & Energy Monitoring)**, a power measurement interface to gather power, voltage, and temperature information as a secondary non-functional measurement trace. Such temperature information is especially important in dependable systems where temperature can also be used as a vector of attack for the system. However, power and temperature collection are extremely hardware dependent, and a consequence is presented in Section 6 of Deliverable D5.2 focusing on the integration of technology bricks to the final iMx8 platform from I&M.

4.8 In-silico AI

During the setup of the platform proposed in D2.1 Section 2.7 several unforeseen challenges were detected. The prior investigation has shown that a sensor fusion approach, with multiple different sensors of different types, is most promising to give good results while being stable. One of the aforementioned challenges is the need for synchronization of all different sensors as well as the limited computational capacity of edge devices like Raspberry Pi, rock Pi X or Jetson Nano. Different sensor types like camera, lidar and radar have different operating conditions like framerate or preprocessing. Hardware accelerators, so called silicon born AI, usually support AI computations that are computationally expensive on normal CPUs, but they are not optimized for the preprocessing of generic data streams as much as they are for image analysis. We performed an in depth investigation about algorithms that need simpler preprocessing so that more processing can be done by in silicon implemented accelerators.

Our results in an approach where preprocessing steps are being handled by the neural network and can be thus be easily run on the accelerators. It is also important to note that preprocessing is highly dependent on the sensor type, as image camera data only requires a normalization, LiDAR data streams require more steps, and the preprocessing of radar data is even more complex. State of the art techniques as well as algorithms designed by us are applied in the system to enable the simultaneous operations on the edge devices.

In the following subsections we will explain in detail the hardware components and their interaction, as well as our radar-based follower algorithm and the interface with the TEACHING software platform to integrate each module in the system.

4.8.1 Physical Prototype Device

The base frame is a 1:10 miniature four-wheel-drive model car. A DC-motor as well as a servo drive, controlling the steering, are mounted on this chassis. To make room for the battery the cardan shaft was taken away, resulting in only the rear wheels being driven. The gearbox was modified to include a cogwheel with magnets and an Infineon TLE4922 Hall sensor to have the possibility of measuring the distance / speed of the model car. This way the motor can be PID-

controlled. The structure carrying everything is 3D-printed. Infineon's micro controller XMC1100 is used to handle all actuators. With the BTN8982 shield the DC-motor is operated.

All sensors are managed with a Raspberry Pi, a Rock Pi X or a jetson nano, depending on the setup. In the following text we will always use the Raspberry Pi as a CPU. As visual input, a 5 MP Raspberry Pi camera, with a viewing angle of 160° is used. Apart from the camera, a LiDAR and a radar sensor are included. The LiDAR sensor is the CUBE1 from Blickfeld with a field of view of 70° x 30°, an angular resolution of up to 0.4° and a range resolution of less than 1 cm. The radar sensor is the 60 GHz BGT60TR13C from Infineon, which is a Frequency Modulated Continuous Wave radar with one transmit and three receive channels with the antenna in package as displayed in Figure 31.

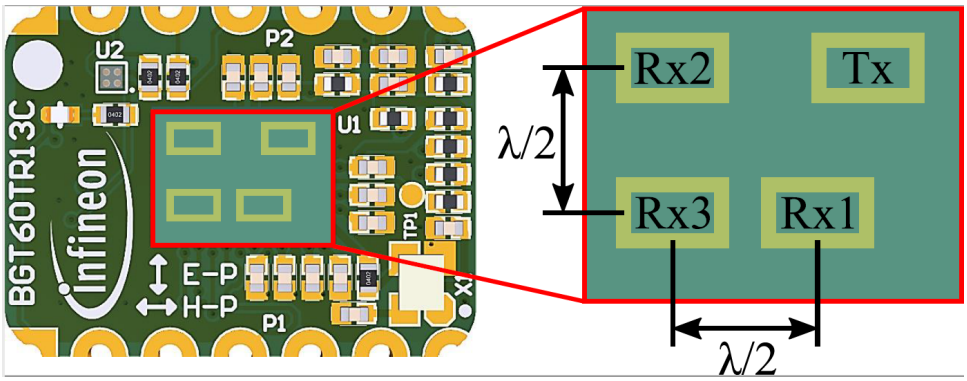


Figure 30: Infineon's BGT60TR13C shield with antenna configuration

Only the two horizontal antennas (Rx3 and Rx1) are used to reduce the amount of data that needs to be processed. The center frequency is 60 GHz, and the bandwidth is set to 4 GHz. Each frame consists of 64 chirps, while for each chirp 128 samples are recorded at a sample rate of 2 MHz.

The connection between the components is illustrated in Figure 32. There are two separate battery packs for power supply, one for all components and an extra one for the LiDAR as the LiDAR operates on a voltage level of 12 V. As the XMC1100 has its own power management system it can be operated directly with the battery while the Raspberry Pi needs an additional DCDC converter. The XMC1100 handles the Hall sensor input and communicates with the Raspberry Pi via I2C to transmit the information of the desired movement as well as to return the data from the Hall sensor. The actuators are controlled directly or indirectly through the XMC1100 by PWM signals.

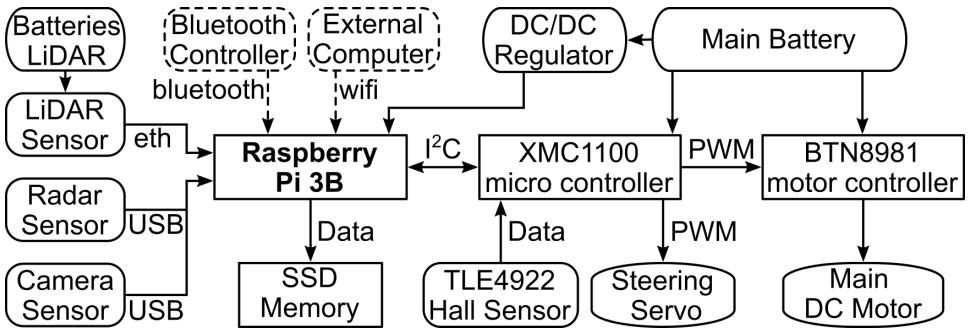


Figure 31: Demonstrator overall design and dataflow diagram

The main controller (here the Raspberry Pi) reads all sensor values, stores them to the SSD and passes orders of how to continue driving to the XMC1100. These are either generated through algorithms evaluating the sensor inputs, or by a human through a Bluetooth controller during the data gathering sessions. The setup and operation of the Raspberry Pi is realized by an external computer connected through WIFI.

Due to the optimization of the hardware communication protocols as well as the libraries to use multiple sensors, the developed system can record data of numerous scenarios using the camera, radar, and LiDAR sensors simultaneously. Each of these sensors have different limitations regarding the data that can measured, i.e., the LiDAR sensor has a physical constraint due to the laser reflection time. This means the LiDAR sensor cannot detect / measure any target closer than 1.5 meters to the sensor. Similarly, the radar sensor, due to the specifications of the sensor, cannot detect targets further than 15 meters, while static targets may be lost in a shorter range.

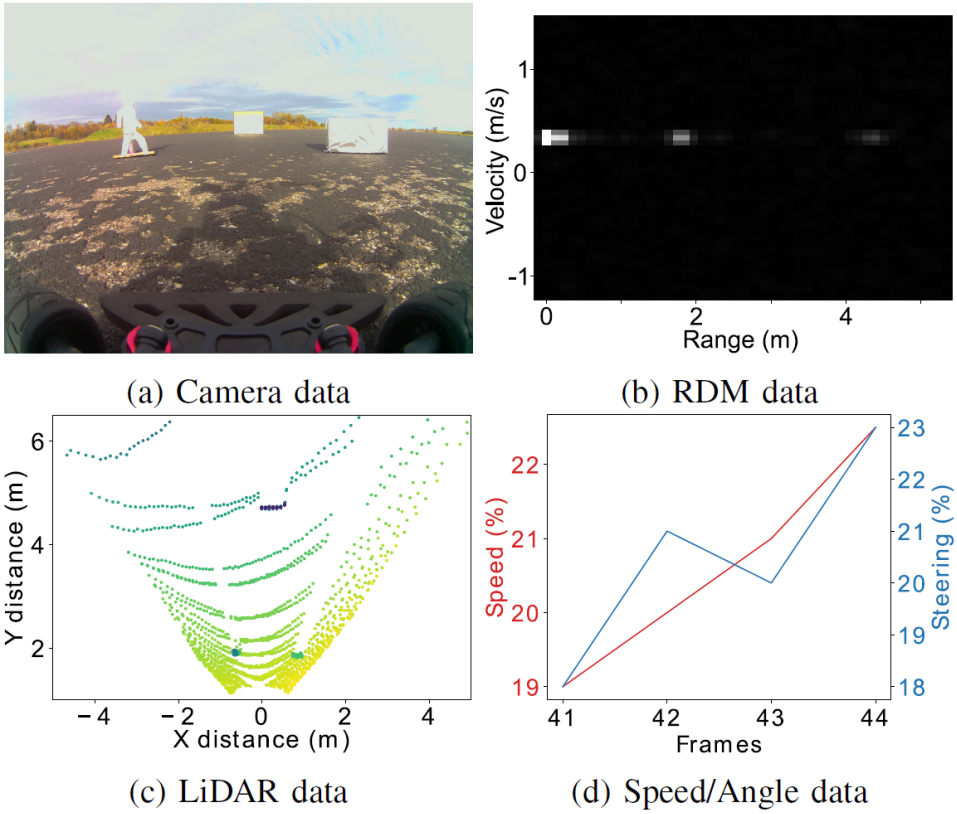
In a future version of the software data streams coming from the different sensors will be published by the prototype via MQTT, exposing the vehicle as just another kind of TEACHING edge device. We plan to exploit this kind of integration with the HPC2I platform both to gather datasets, that can be reused off-line by all project partners, and to provide real time access to the data for other research applications, e.g., to compare the on-board AI results with those computed by more powerful resources in the global computing platform.

4.8.2 Experimental Settings and Dataset Gathering

Multiple scenarios were tested with this system in order to ensure its proper functioning under different circumstances, namely road intersection, roundabout, tight streets and wide streets. At the same time, different targets were used during the data recording sessions to determine the minimum size of the targets, leading to a final decision of using targets of, at least, 10x10 centimeters.

As a result of this, different datasets were created following a structure where the information of the model car, the radar data, the LiDAR data and the camera data is saved in a synchronous approach. The frame rates of the sensors have been configured based on the slowest sensor leading to a frame rate up to 20 frames (fps). The data from the internal state of the car (speed, steering, timestamp) is stored as a JSON file, the camera sensor data is saved as JPG images, the LiDAR data is stored as 3D points, that later can be used for multiple representations such as top view or depth map, and finally the radar data is stored as raw data that can be later used to generate Range-Doppler-Maps (RDM) as shown in Figure 32.

Figure 32: Frame sample of camera, radar, LiDAR and Speed/Steering data.



(a) Camera data (b) RDM data

(c) LiDAR data (d) Speed/Angle data

The described platform is used to record datasets on a fixed track that is specified in the following. During the recording the vehicle moves with an average speed of 253 mm/sec.

A total of 153 648 frames are recorded during 50 laps. The dataset is split in a validation subset (15 115 frames) and a train/test one (138 533 frames). The radar data as well as the user input are also stored.

The quality of a Neural Network (NN) is, apart from its architecture, highly dependent on the amount and the quality of the data it is trained with. In our in-depth analysis of the "follower" use-case ideally all possible curvatures to both sides, as well as straight driving and different vehicle speeds should be used for learning. On these grounds we designed a track shown in Figure 33, which in theory has two curvatures that represent 10 meters (38.5 %) each while the part of the track that goes straight aggregates to 6 meters (23 %). This way both curvatures are evenly represented in the recorded data set that is used for training.

The training data is gathered by maneuvering the vehicle by means of a remote controller, performing various laps through the track while recording the user input and the sensor signals. The histogram in Figure 34 shows the distribution of the human steering input, for various laps through the track. The normalized steering value ranges from -1 to 1, representing the deflection of the front wheels. It can be noticed that the steering is evenly distributed among both left and right bendings, though the curvatures do not show only one deflection (as one could assume due to the fixed curvatures bends of the track) but rather smooth transitions.

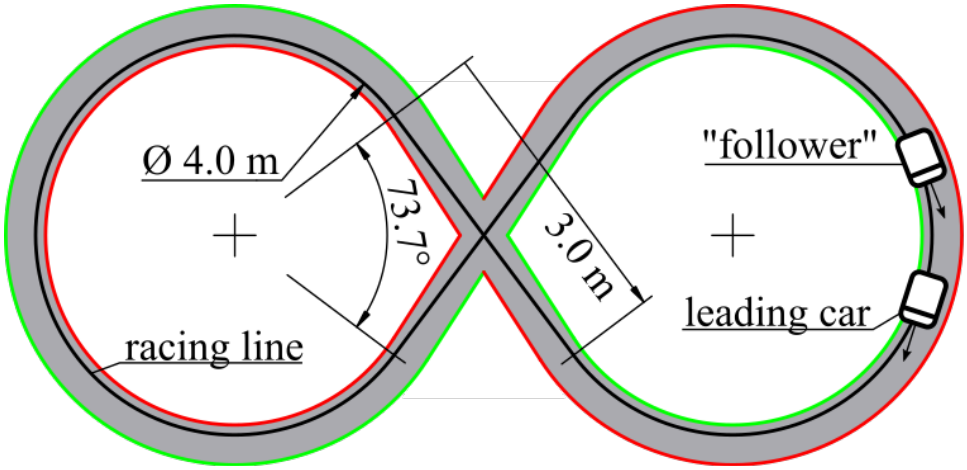


Figure 33: Layout of the training data track

The underrepresentation (14 % instead of 23 %) of straight driving which is indicated by a steering value of 0.0 is a result of faster driving during the straight part of the track, as well as minor corrections in heading during the straight driving part of the track.

To be able to autonomously maneuver the miniaturized vehicle in the "follower" use case, we have to extract well-defined information streams from the radar data: distance, speed, angle and heading of the preceding vehicle. The autonomous control based on a DCNN navigates the platform by the output coming from the radar pipeline.

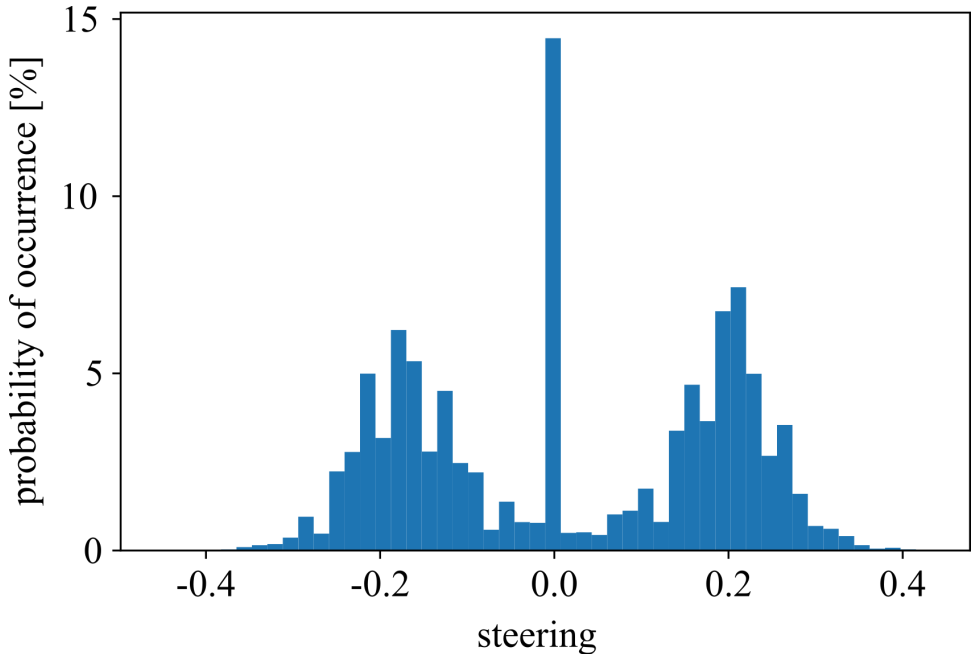


Figure 34: Histogram of the human controlled steering input during training. Data are collected over a total of 138 533 frames during 46 laps. Here -1 and 1 represent the maximum deflection of the steering towards the left and the right side

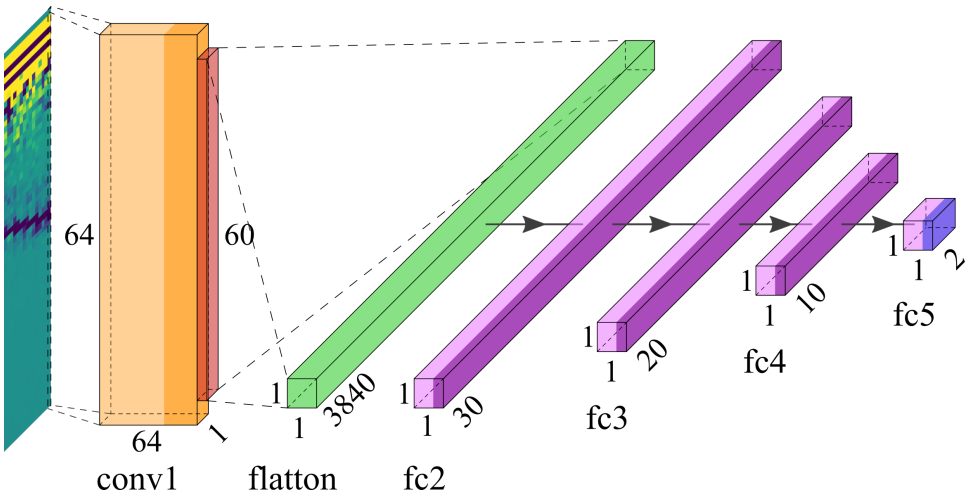


Figure 35: Architecture of the deep convolutional neural network

4.8.3 AI-based Processing of Radar Data

The raw radar data gathered with a FMCW-radar is structured as a three-dimensional data cube for each frame, where the dimensions represent

1. the number of antennas
2. the number of chirps per frame and
3. the number of samples per chirp.

Traditional approaches for radar processing make use of the Fast Fourier Transformation (FFT) over the fast time dimension to get a map of the intensity over the range, followed by a second FFT in the slow time dimension, which results in a Range Doppler Map (RDM) for each antenna, representing the complex-valued velocity in the different range bins for each antenna. From the RDMs of antennas that are oriented in a $\lambda / 2$ distance horizontally or vertically the azimuth as well as the elevation of localized targets can be calculated and thus the Range Angle Map (RAM) is constructed. In our use case we limit the number of available targets in the radar's field of vision to one, being the preceding vehicle.

To reduce the amount of data forwarded to the neural network and increase the processing speed, we do not calculate the Range Angle Map occupation, which would be the traditional approach, but we introduce a new method of only considering a subset of a covariance from the raw data cube.

The selected DCNN design is shown in Figure 35. The first layer is a 1D-convolution (*conv*) layer to cope with the effects originating from the quantization of the samples for each chirp that result in the range bins. We chose a kernel of size 5 and a stride of 1. No padding is applied and the number of filters is chosen to be 64. After the initial convolution layer its output is flattened. The following fully connected (*fc*) layers learn the linkage between the extracted features from our optimized input and the outputs needed for the autonomous control of the vehicle. Each layer, be it conv or fc, is followed by a rectified linear unit activation function. Finally, the dense output layer is performed with a linear activation function.

A snippet of the gathered data from one entire lap through the track is shown in Figure 36 below. The upper part represents our optimized input, where the range bins reflect the distance to the preceding vehicle. The color is the visual representation of the value in each bin, depicting the relative angle of the received radar signal in each bin. In the lower part the corresponding input from the human operator during the recording is depicted. The correlation between steering and the angle information in the radar signal, as well as the distance of the main reflection and the throttle is shown.

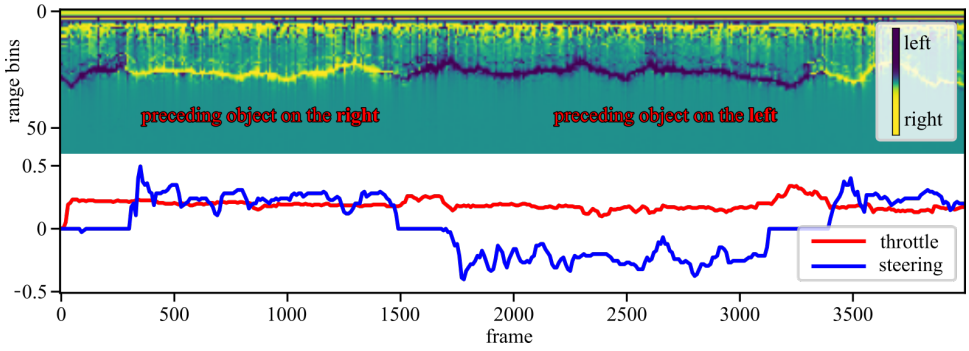


Figure 36: The correlation between human input and radar data.

The final size of the model (h5 version) with 116 466 trainable parameters is 1.4 MB which is small enough to be running on the Raspberry Pi under the given timing constrains.

Training of the model with the prior described dataset takes about 50 seconds on an office laptop for each epoch. The trained model is evaluated on the validation subset where the networks prediction deviates by 0.038 on average for the steering and by 0.008 for the throttle. Evaluation of the model with the demonstrator and not pre-recorded data gives satisfying results.

The trained model reaches its limits if the leading vehicle operates outside the trained conditions, especially smaller turns than the 4 meters in diameter lead to unexpected behaviour of the "follower". The utilization of the processors is on average about 85% at a processing frame rate of 20 Hz.

The very low deviation of the throttle compared to the steering can be explained by the fact that during the recording as well as during testing the speed was always more or less constant. Starting and stopping as well as accelerations were not evaluated. Another simplification was a speed invariant distance control as well as the assumption of exactly one target being present in the range of the radar.

4.8.4 Future Extension and Integration within TEACHING's HPC2I

As discussed above, future experiment recordings will include several different scenarios, allowing to train the system to a broader set of conditions. As mentioned in Section 4.8.1, interfacing the data collection via MQTT will allow live collection of data within the HPC2I architecture, allowing for

- offline dataset building as well as
- performing live training on the data via HPC2I-owned devices and comparison of on-device results with off-device ones.

A future task with respect to the embedded system is to further optimize the pre-processing of the radar data, in order to further reduce processor utilization.

Finally, the security of the radar data and its evaluation algorithms will be tested regarding possible threats like adversarial attacks or jamming, as the radar sensor data is safety critical in this scenario, especially if applied in a future, real sized vehicle.

A crucial step is the in-silicon implementation of the presented AI algorithms, as well as the provision of these computational unit throughout the entire platform. The development process of the neural networks in these experiments is happening with the focus and the intention to run them in an in-silicon implementation accelerator. The specifications under which the architecture of the AI component is designed match the specifications of a custom hardware designed to support in-silico AI implementation. The hardware is designed in another research project that Infineon participates to, thus also representing an opportunity for collaboration. A schematic of its architecture is shown in Figure 37.

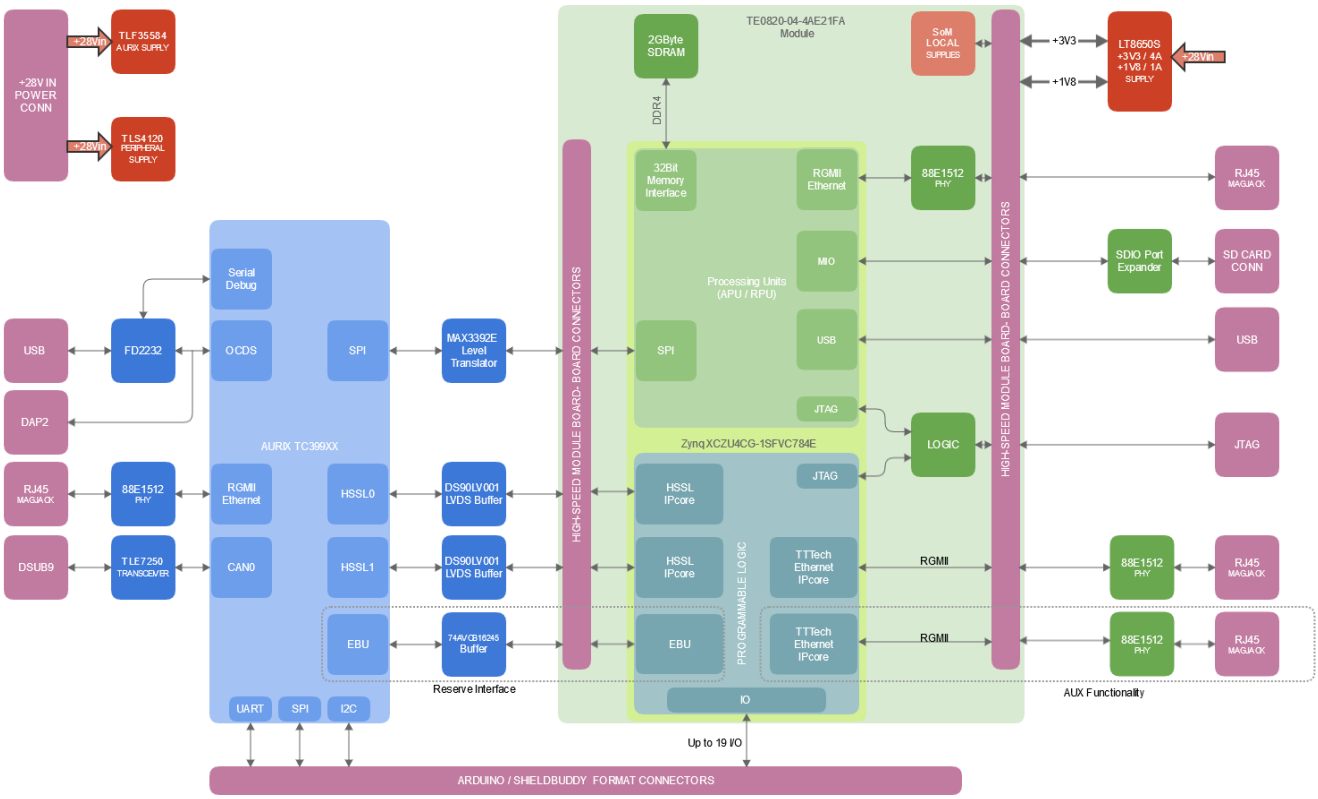


Figure 37: Supporting HW Schematics for In-Silicon Experiments

The design exploits the interaction of an Infineon Aurix device with a Xilinx device to realize an in-silicon implementation of a tensor-based accelerator for AI. The layout of the PCB, that we show in Figure 38, is also to be header compatible with Arduino specifications, supporting a broad span of options for interfacing and device integration.

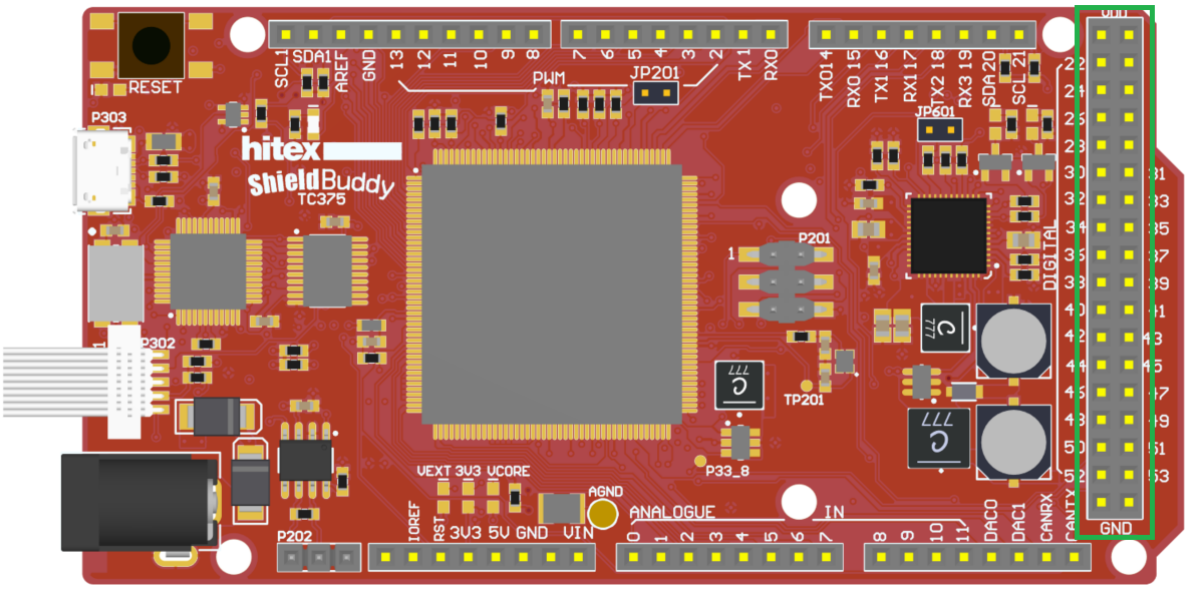


Figure 38: PCB layout of the Aurix - Xilinx in silicon board.

5 Conclusions (CNR)

This deliverable describes in a consolidated way the results achieved by WP2 in its development process, devoted to realizing the HPC2I.

To this end, D2.2 provides an update (with respect to D2.1) to the state-of-the-art analysis (where needed) focused on the research and development endeavors conducted by the different tasks belonging to the work package.

Furthermore, the deliverable refines the architecture presented in D2.1 by tailoring it to the actual technologies adopted to realize the components of HPC2I. Concerning such components, their prototypes are presented and discussed while motivating the implementation choices, including the rationale guiding the selection of base models and technologies.

As a matter of fact, this deliverable describes an interim version of the HPC2I, that is the basis on top of which to build the final version of the TEACHING communication and computing infrastructure.

6 Appendix: Kafka/MQTT Bridge Examples

6.1 The MQTT Library

The MQTT protocol plays a role in the HPC2I architecture as a low overhead pub/sub mechanism that can work on the network but is more suited to local use and allows easily interfacing to the more powerful Kafka pub/sub network.

Note that the example in this section (6.1) is very similar to the description of the MQTT usage done in deliverable D4.2, where MQTT is the key mechanism for locally routing data streams within an AIaaS instance.

Data sending and receiving, which will be done through the broker, happen throught he functions in the MQTT library, as shown in the following.

The subscribe method accepts 2 parameters: a topic or topics and a QoS (Quality of Service) with conventional values within [0 - 2].

Method 1- Uses a single topic string

```
client1.subscribe(car/tyres/tyrepressure1,1)
```

Method 2- Uses single tuple for topic and QoS -(topic,qos)

```
client1.subscribe((car/tyres/tyrepressure2,2))
```

Method 3- Used for subscribing to multiple topics using a list of (topic, qos) tuples

```
client1.subscribe([(car/tyres/tyrepressure1,2),
      (car/tyres/tyrepressure3,1),(car/tyres/tyrepressure4,0)])
```

The subscribe function returns a tuple to indicate success as well as a message id which is used as a tracking code.

```
(result, mid)
```

The MQTT broker/server will acknowledge subscriptions which will then generate an *on_subscribe* callback. The format of the callback function is shown below:

```
on_subscribe(client, userdata, mid, granted_qos)
```

The *mid* value can be compared with the one returned by the function call to check for successful subscription requests.

The publish method accepts 4 parameters.

```
publish(topic, payload, qos, retain)
```

The *payload* is the message you want to publish, the *topic* is where you want to publish, the *qos* is the reliability (QoS) for the sent messages, the flag *retain* when set to "on" allows the broker to store the last message and the corresponding QoS for that topic.

```
client.publish("car/light","ON",1,off)
```

Before a client to start publishing or subscribing operation, it needs to create a client instance, connect to a broker.

```
broker_address="192.168.1.184"
client = mqtt.Client("P1") #create new instance
client.connect(broker_address) #connect to broker
client.publish("car/tyres/tyrepressure1","LOW")
client.subscribe("car/tyres/tyrepressure1")
```

6.2 Exploiting the Kafka Paradigm within the HPC2I Architecture

In the following example, we show how to code in Python a producer that emits numbers from 1 to 1000 and sends them to our Kafka broker. Then we show the code of the corresponding consumer, that will read the data from the broker and store them in a MongoDB collection.

The advantage of using Kafka is that, if our consumer breaks down, the new or fixed consumer will pick up reading where the previous one stopped. This is a great way to make sure all the data is fed into the database without duplicates or missing data.

6.2.1 Data producer code

Create a new Python script named producer.py and start with importing time.sleep, json.dumps, and kafka.KafkaProducer from our the Kafka-Python library.

```
from time import sleep
from json import dumps
from kafka import KafkaProducer
```

Then initialize a new Kafka producer. Note the following arguments:

- **bootstrap_servers=['localhost:9092']** sets the host and port the producer should contact to bootstrap initial cluster metadata. The example shows the default values, so in this case we could leave out the parameter and still get localhost port 9092.
- **value_serializer=lambda x: dumps(x).encode('utf-8')** is the function performing data serialization before data is sent to the broker. Here the data is converted to a json file and encoded to utf-8.

```
producer=KafkaProducer(bootstrap_servers=['localhost:9092'],
                value_serializer=lambda x:
                dumps(x).encode('utf-8'))
```

Now, we want to generate numbers from one till 1000. This can be done with a for-loop where we feed each number as the value into a dictionary with one key: number. This is not the topic key, but just a key of our data. Within the same loop we will also send our data to a broker.

This can be done by calling the send method on the producer and specifying the topic and the data. Note that our value serializer will automatically convert and encode the data. To conclude our iteration, we add a 5 second pause. To actually make sure the message is received by the broker, it's best to also register a callback, a step which is not shown here for the sake of simplicity.

```
for e in range(1000):
      data={'number' : e}
      producer.send('numtest', value=data)
sleep(5)
```

6.2.2 Data consumer code

Before we start coding our consumer, create a new file consumer.py and import json.loads, the KafkaConsumer class and MongoClient from pymongo. The mongo code can be replaced with any other code. This can be code to feed the data into another database, code to process the data or anything else. For more information about PyMongo and MongoDb, please consult the documentation.

```
from kafka import KafkaConsumer
from pymongo import MongoClient
from json import loads
```

Let's create our KafkaConsumer and take a closer look at the arguments:

- The first argument is the topic, 'numtest' in our case.
- bootstrap_servers=['localhost:9092']: the same as our producer.
- auto_offset_reset='earliest': it handles where the consumer restarts reading after breaking down or being turned off and can be set either to earliest or latest. When set to latest, the consumer starts reading at the end of the log. When set to earliest, the consumer starts reading at the latest committed offset.
- enable_auto_commit=True: makes sure the consumer commits its read offset every interval.
- auto_commit_interval_ms=1000ms: (not shown in the example) sets the interval between two commits. Since messages are coming in every 5 s., committing every second seems fair.
- group_id='counters': this is the consumer group to which the consumer belongs. A consumer needs to be part of a consumer group to make the auto commit work.
- The value_deserializer function parameter in the example deserializes the data into a common json format, the inverse of what the value serializer does.

```
consumer = KafkaConsumer(
    'numtest',
    bootstrap_servers=['localhost:9092'],
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    group_id='my-group',
    value_deserializer=lambda x: loads(x.decode('utf-8')))
```

The code below connects to the numtest collection (a collection is similar to a table in a relational database) of our MongoDb database.

```
    client = MongoClient('localhost:27017')
    collection = client.numtest.numtest
```

We can extract the data from our consumer by looping through it (the consumer is an iterable). The consumer will keep listening until the broker doesn't respond anymore. A value of a message can be accessed with the value attribute. Here, we overwrite the message with the message value. The next line inserts the data into our database collection. The last line prints a confirmation that the message was added to our collection. Note that it is possible to add callbacks to all the actions in this loop.

```
    for message in consumer:
        message = message.value
        collection.insert_one(message)
        print('{} added to {}'.format(message, collection))
```

The learning module in the core can communicate through the infrastructure using the MQTT protocol. It is also possible to use the Kafka Protocol, as shown in Figure 22 on the right. Having the learning modules communicating with two different protocols is in principle redundant. We keep this option open as it may allow dynamic Kafka integration in the core of the platform of the modules developed in WP4, to allow for specific tasks that go beyond the use in a single CPS. Note that communicating MQTT the remote learning module or its proxy component in the AIaaS architecture can use the functions described in Section 4.5.1.4 and Section 6.1. For the data sending/receiving, which will be done through the broker, they will simply use the functions made available by the library.

A Kafka Connect source connector reads data from MQTT's topics and publish them to Kafka ones as shown in the following:

```
INSERT INTO name topic in Kafka SELECT * FROM name of the MQTT topic
       [WITHCONVERTER=myclass]
```

where we selected all the fields of the MQTT topic.

Before the connector can start publishing data on Kafka it needs to execute the following initialization operations:

```
name=mqtt-source
connector.class=com.datamountaineer.streamreactor.connect.mqtt.source.Mqt
tSourceConnector
tasks.max=1
connect.mqtt.kcql= INSERT INTO name topic in Kafka SELECT * FROM name of
the MQTT topic
WITHCONVERTER=`com.datamountaineer.streamreactor.connect.converters.sourc
e.JsonSimpleConverter`
connect.mqtt.client.id=dm_source_id
connect.mqtt.hosts=tcp://mqtt:1883
connect.mqtt.service.quality=1
```

A Kafka Connect sink connector reads data from Kafka's topics and publishes them to MQTT ones by setting the KCQL property at creation time, as shown in the following.

Analogously, before the connector can start publishing data on MQTT it needs to execute the following initialization operations.

```
name=mqtt
connector.class=com.datamountaineer.streamreactor.connect.mqtt.sink.MqttS
inkConnector
tasks.max=1
topics=orders
connect.mqtt.hosts=tcp://mqtt:1883
connect.mqtt.clean=true
connect.mqtt.timeout=1000
connect.mqtt.keep.alive=1000
connect.mqtt.service.quality=1
connect.mqtt.client.id=dm_sink_id
connect.mqtt.kcql= INSERT INTO MQTT_Topic SELECT * FROM Kafka_topic
```

This sequence inserts into the *MQTT_Topic* all fields from *Kafka_topic*

```
INSERT INTO MQTT\_Topic SELECT * FROM Kafka\_topic
```

while this sequence inserts into *MQTT_Topic* all fields from *Kafka_topic* from dynamic field *field*

```
INSERT INTO `$field` SELECT * FROM Kafka_topic WITHTARGET = field
```

In the last example, the sink connector can dynamically write to MQTT topics determined by a field in the Kafka message value, by using the *WITHTARGET* clause and specifying *field* as the target in the *KCQL* statement.