Accelerating intelligent vehicle vision: A hybrid Python-Rust architecture with partial-blocking inter-process communication

Dávid SZILÁGYI, Răzvan FILEA, Kuderna-Iulian BENȚA

Faculty of Mathematics and Computer Science, Babes-Bolyai University, Cluj-Napoca, Romania

davtszi@gmail.com, razvan.filea@stud.ubbcluj.ro, kuderna.benta@ubbcluj.ro

Abstract: Modern autonomous systems require high-throughput, reliable vision pipelines with real-time guarantees. This paper presents a hybrid Python-Rust architecture that leverages synchronized shared memory and introduces a novel partial-blocking inter-process communication (IPC) model to decouple perception modules and enable parallelism without GIL-induced bottlenecks. In order to ensure a consistent data handling across asynchronous pipelines, the proposed system embeds lightweight runtime verification through watchdogs and health monitoring, and avoids redundant memory copying through efficient shared buffers. Across embedded and desktop platforms, this system delivers a fourfold speedup over naive multiprocessing while reducing the memory and processing overhead. The system is evaluated under various runtime configurations and demonstrates real-world applicability on a 1:10 scale autonomous vehicle. Its architecture provides a scalable foundation for safety-critical, real-time perception pipelines for robotics applications.

Keywords: Intelligent Vehicles, Real-Time Vision, Shared Memory IPC, Python-Rust Integration, Partial-Blocking Communication.

Accelerarea viziunii inteligente a vehiculelor: o arhitectură hibridă Python-Rust cu comunicare inter-procese cu blocare parțială

Rezumat: Sistemele autonome moderne necesită pipeline-uri de vizualizare fiabile, cu debit ridicat și garanții în timp real. Acest articol prezintă o arhitectură hibridă Python-Rust care utilizează memorie partajată sincronizată şi introduce un model nou de comunicare între procese (IPC) cu blocare parțială, pentru a decupla modulele de percepție și a permite paralelismul fără blocaje induse de GIL. Pentru a asigura o gestionare consistentă a datelor în pipeline-urile asincrone, sistemul propus include verificări ușoare la rulare prin watchdog-uri și monitorizarea stării de sănătate (se referă la starea internă a sistemului, funcționarea componentelor, detectarea erorilor sau a problemelor înainte să afecteze procesul), evitând totodată copiile redundante de memorie prin utilizarea eficientă a bufferelor partajate. Pe platformele integrate şi desktop, acest sistem oferă o accelerare de patru ori mai mare față de multiprocessing-ul simplist, reducând totodată consumul de memorie și suprasarcina de procesare. Sistemul este evaluat în diverse configurații de rulare şi demonstrează aplicabilitate în condiții reale pe un vehicul autonom la scară 1:10. Arhitectura oferă o bază scalabilă pentru pipeline-uri de percepție în timp real, critice pentru siguranță, în aplicații de robotică.

Cuvinte-cheie: Vehicule inteligente, Viziune în timp real, IPC cu memorie partajată, Integrare Python-Rust, Comunicare cu blocare parțială.

1. Introduction

Advanced driver-assistance systems (ADAS) and autonomous cars (Xu, Zhang & Sun, 2025) base their efficiency on high-performance perception modern autonomous systems. The systems must process huge volumes of sensor data in real time continuously to ensure accurate situational awareness, and as a result, the computational efficiency and architectural scalability are the overriding design priorities (Rosique et al., 2019). Consequently, the performance or latencies in these pipelines can have a direct impact on their safety, reliability, and responsiveness, especially in dynamic, high-risk environments like in ADAS.

https://doi.org/10.33436/v35i3y202508

While building such systems one has to overcome a number of challenges. Developers must integrate computer vision algorithms, machine learning models. Hardware-specific optimizations within a cohesive software architecture should support modular development, real-time performance, and cross-platform compatibility. Moreover, the increasing complexity of the perception tasks necessitates a collaborative and scalable development model. Different modules - from low-level data acquisition to high-level semantic interpretation - should be independently developed, tested, and deployed.

However, it is well known that the conventional single-language systems often fail to balance rapid prototyping needs with runtime efficiency. A well accepted solution, Python, while dominant in the AI and CV ecosystem due to its flexibility and extensive libraries, suffers from: 1) Global Interpreter Lock (GIL) constraints and, 2) limited concurrency on processing bound tasks. Meanwhile, low-level systems languages like Rust offer deterministic performance and safety but lack the expressive tooling ecosystem for vision development.

This paper proposes an architecture that is concurrent, allows for a modular vision and tightly integrates Python-based high-level Computer Vision (CV) pipelines with a performant Rust backend for data transport and synchronization. In its design the common IPC bottlenecks are eliminated by using synchronized shared memory and a novel partial-blocking communication strategy is introduced. This allows faster components to progress without waiting for slower modules. This is an essential feature for the real-time systems where different tasks vary significantly in complexity.

The remainder of this paper is organized as follows: Section 2 reviews related work across perception systems, middleware frameworks, and multi-language integration. Section 3 outlines the architectural requirements and introduces the Producer-Consumer communication strategies, including the proposed partial-blocking approach. Section 4 details the methodology, pipeline structure, concurrency design, and the *SharedMessage* specification with its implementation variants. Section 5 presents the evaluation, including benchmarks, hardware-software testing, operational modes, and a real-world deployment on a 1:10 scale autonomous vehicle. Section 6 discusses limitations and future directions, the final section concludes with key findings and contributions.

2. Related work

Real-Time (RT) Computer Vision architectures for autonomous vehicles have been widely investigated, particularly with respect to modularity, throughput, and robustness. Below, there are reviews of the relevant contributions across perception systems, middleware frameworks, multi-language integration, and runtime safety.

Perception Systems and Sensor Fusion. Rosique et al. (2019) provide a comprehensive survey of the perception systems, sensor modalities, and simulation tools for autonomous vehicles. The previous works (Jahromi, Tulabandhula & Cetin, 2019; Gu, Wang & Qin, 2021) proposed real-time, low-latency architectures for multi-sensor systems. The importance of the hybrid offline-online paradigms for enhancing the system reliability was depicted in Ros et al. (2015).

Vision Architectures and Middleware. In the early embedded CV systems (Sridharan & Stone, 2005), the latency-performance tradeoffs in resource-constrained environments were discussed. Middleware frameworks like ROS2 (Macenski et al., 2022) and visualization tools like RViz (Kam et al., 2015) demonstrate the transition toward distributed, modular runtime systems with built-in support for visualization and debugging.

Multi-Language Integration and IPC. The integration of Python and Rust through PyO3 (Flitton, 2022) supports the hybrid architectures that combine development flexibility with high-performance execution. The foundations for the shared-memory synchronization, which are critical for the present partial-blocking IPC design was presented in Scott & Brown (2013).

3. Architecture requirements

To design an effective RT CV architecture specifically for autonomous vehicles necessitates addressing several critical requirements to ensure both high performance and developer accessibility.

Foremost, the system must operate in both RT and offline modes. RT processing is critical for in-vehicle deployment, where timely responses are critical (Jahromi, Tulabandhula & Cetin, 2019; Gu, Wang & Qin, 2021). The offline capability, on the other hand, enables efficient debugging, development, and testing using pre-recorded data streams (Ros et al., 2015).

Secondly, the integration with the Python programming language is essential. Python continues to be a cornerstone of the modern CV development, thanks to its extensive ecosystem of libraries such as OpenCV (Bradski, 2000), TensorFlow (Abadi et al., 2016), and PyTorch (Paszke. et al., 2019). Python-based workflows should be supported for the architectures used for research and rapid prototyping. Thirdly, the system to be must support the efficient handling of video data. This includes the ability to save both raw input and processed output streams with minimal performance overhead - critical for debugging, post-run analysis, and RT performance evaluation.

Finally, RT visualization is vital for monitoring system's behaviour both during development and deployment. Widely adopted tools like Rviz (Kam et al., 2015), integrated within ROS2 (Macenski et al., 2022), exemplify the importance of live visualization for system's validation and user assurance.

Together, all these requirements define a flexible and performant architecture capable of supporting RT perception workloads not only in experimental but also in operational settings.

3.1. Producer-Consumer model

The Producer-Consumer model is a fundamental element of any RT vision architecture. It reigns the flow of data from a producer to one or more consumers. This proposed model supports three distinct strategies: standard Non-Blocking (NB), Full-Blocking (FB), and the newly proposed Partial-Blocking Approach (PBA).

In the NB strategy, the *Producer* continuously publishes new frames without waiting for *Consumers* to finish processing. *Consumers* operate independently. Each of them processes the most recent data available in its own rhythm. While this maximizes the throughput, it may also cause synchronization issues, as *Consumers* might work on different frames simultaneously. The method described here is well-suited for tasks where responsiveness outweighs consistency, such as the RT monitoring.

The FB strategy enforces synchronization by requiring the *Producer* to wait until all *Consumers* have processed the current frame before publishing the next one. Although this guarantees consistency across all *Consumers*, it introduces latency and reduces the throughput. This approach is best suited for pipeline development and testing scenarios, where the uniform frame handling is critical.

The present PBA strategy reaches a fair balance between these extremes. The *Producer* may release new data as soon as at least one *Consumer* has begun processing the previous frame. This improves the throughput by eliminating the idle time, ensuring that every frame is processed by at least one *Consumer*. It is particularly effective for the RT systems like the autonomous driving, where fast, safety-critical tasks must run concurrently with slower planning or logging modules.

With the combination of these three strategies, the Producer-Consumer model offers a flexible mechanism for managing data flow, enabling the architecture to well balance consistency, responsiveness, and performance based on the current application's needs.

4. Methodology

4.1. Pipeline architecture

To organize the vision processing into modules a sequential pipeline structure, as originally presented by Meunier (1995) is augmented to fit the proposed requirements. The raw frame, as well as any processed frames produced so far and the extracted features, like bounding boxes or lane parameters, are combined into a PipeData object, that also features a timing component to track per module performance.

Then a Filter is used to represent a single processing step. Whether performing region-of-interest extraction, grayscale conversion, edge detection, or deep-learning-based object detection, upon invocation, the filter reads fields from the input PipeData, applies its algorithm to the current frame, and returns a new PipeData that updates the processed-frame field and appends any newly extracted features. The system should remain highly extensible. It may be done by isolating each processing step as a *Filter* and encapsulating both frames and features within PipeData.

In the simplest, fully sequential setup, filters are chained one after another: the raw frame enters *Filter 1*, its output feeds *Filter 2*, and so on. Downstream components - such as decision-making or visualization - receive the final PipeData after the last filter completes. Optionally, an intermediate saving module can record either the raw input or the fully processed frame at any point in this processing chain.

Figure 1 illustrates the sequence of Filters used as the baseline sequential design in these experiments. It features an initial frame capture step, followed by transformation and processing steps and ending in visualization, for a real-time inspection of the detected features, and saving.

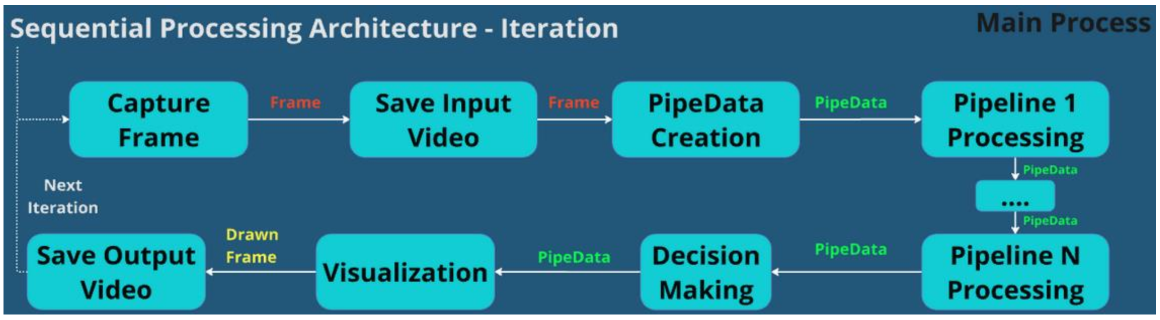

Figure 1. Sequential Processing Architecture

4.2. Concurrency and parallelism

The sequential design is conceptually simple: RT perception pipelines for intelligent vehicles typically execute multiple tasks-lane detection, traffic sign detection, traffic-light recognition, and pedestrian detection - in parallel on each frame. Figure 2 shows this pattern: a single raw frame is dispatched into four branches, each performing cropping and a specialized filter cascade. The individual PipeData outputs are merged before the downstream decision-making.

In Python, achieving true parallelism for Central Processing Unit (CPU)-bound filters is hindered by the Global Interpreter Lock (GIL), which serializes the bytecode execution within a single process. Although multithreading can interleave I/O or Graphics Processing Unit (GPU)-bound operations, it cannot fully exploit multicore CPUs for Python filter logic. Consequently, each branch must run in its own process to bypass the GIL.

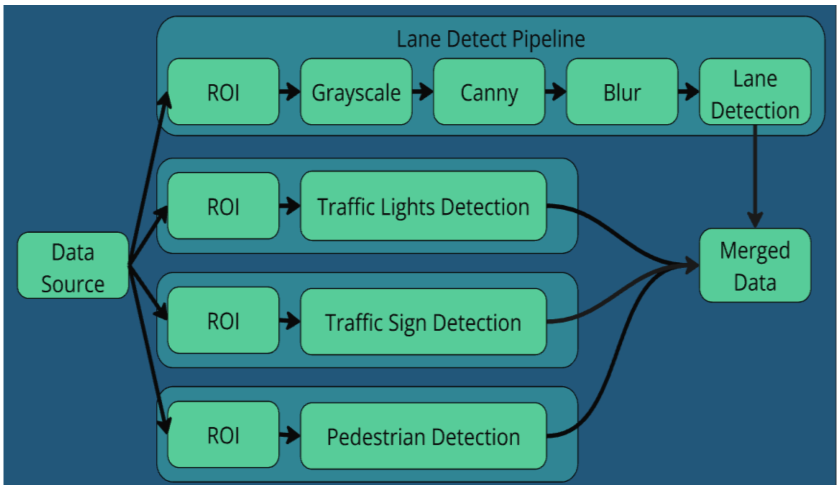

Figure 2. Simple Autonomous Driving Vision Pipeline

A naive multiprocessing implementation, exemplified in Figure 3, would serialize the raw frame and send a copy to each worker via pipes. Each worker deserializes, executes its filter chain, then serializes the updated PipeData back to the manager. This design, however, incurs two key drawbacks: (1) every frame is copied and serialized for each branch, and (2) the overall framerate is limited by the slowest branch. When one branch takes 40ms to process while others take 10ms, iteration time is 40ms with more efficient branches waiting idly for most of each cycle.

This design yields high overhead, especially with large PipeData objects containing high-resolution frames and greater than one extracted feature based on the large copying, serialization, and deserialization of the same data between different branches that is memory and time inefficient.

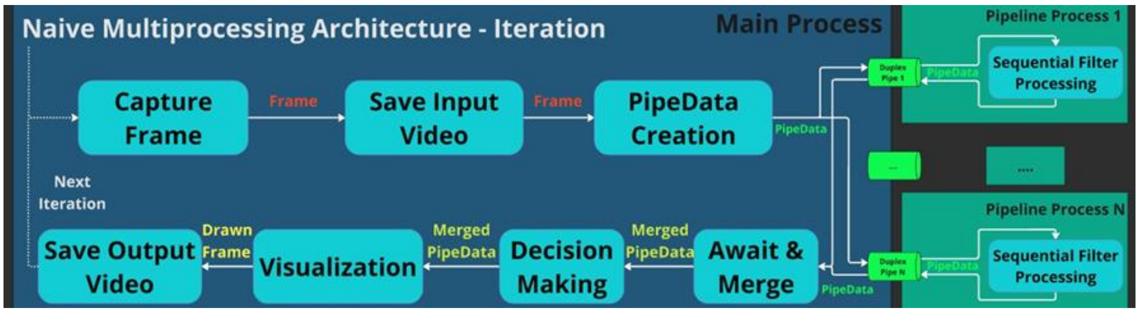

Figure 3. Naive Multiprocessing Architecture

4.3. Proposed SharedMessage based architecture

In order to handle the IPC overhead and ensure runtime longevity, the authors employ an array of shared-memory buffers rather than per-branch pipes. The manager allocates a shared-memory space of a size sufficient for a single PipeData object as input to the pipeline. As shown in Figure 4, worker processes map this space during the startup, read the raw frame in place, and store its filter outputs (processed frame slice and feature data) into the corresponding shared memory slot. The manager can then read at its convenience the streams of the latest PipeData from any worker, aggregate the results, and push them downstream to visualization or decision-making components. All in all this architecture entails N + 2 shared memory buffers: one for the raw frame, one for each output of a worker, and one for the manager to broadcast the combined results, which is a significant reduction from 2N + 2 pipes in the naive approach.

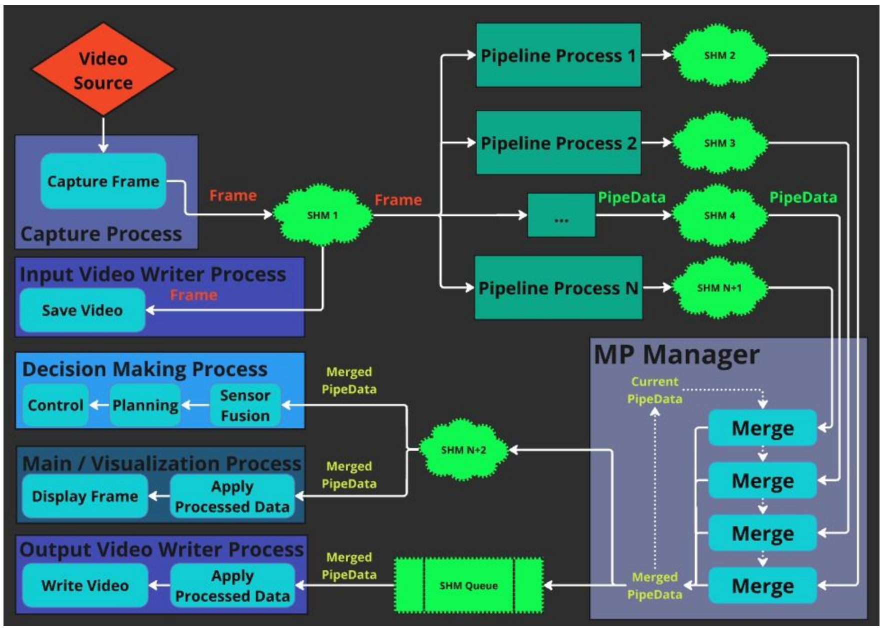

Figure 4. *SharedMessage* Architecture

Moreover, a naive approach lacks runtime guarantee: if a worker is hung or crashed half-way through the process, the manager will block forever waiting for its result. To satisfy the runtime verification requirements, all processes need to provide health monitoring and error recovery hooks. In the present design, each worker process spawns a background "heartbeat" thread which outputs a timestamp every so often to a special shared memory queue. The manager monitors the heartbeats: if the timestamp of a branch does not change over a timeout, the manager designates it as "unresponsive" and may induce a pre-configured recovery action (restart the branch execution or substitute a safe default output). Through embedding monitoring and synchronized shared memory IPC in every branch, the system achieves high throughput and runtime guarantee: the speedy branches execute ahead without waiting for the sluggish ones, and any fault in a branch can be contained, diagnosed, and mitigated in real-time.

4.3.1. Shared memory and Python

Although Python's multiprocessing module offers a basic shared memory API, it lacks the necessary synchronization primitives to enforce consistent, race-free updates by multiple processes. Specifically, there is no simple way to atomically check or update version counters without holding a lock, and Python's combined lock/condition-variable abstraction forces all waiters to wake on any state change, reducing the efficiency.

To get over these shortcomings, a higher-level *SharedMessage* abstraction with explicit versioning, an independent atomic version counter, and dedicated mutex and condition variables is employed. The abstraction offers integration with all three modes of communication: non-blocking, partial-blocking, and full-blocking (as brought in Section 3.1).

4.4. The SharedMessage specification

At its core, the SharedMessage consists of the following:

- A fixed-capacity *Message* region of size C bytes, allocated at startup;
- A 64-bit *MessageVersion* counter (an atomic integer) incremented on each write;
- A 32-bit *MessageSize* denoting the number of valid bytes in the current message;
- A 32-bit *ConsumerCount* indicating how many reader processes exist;
- A 32-bit *ReadCount* that tracks how many readers have consumed the current version;

- A flag to signal when the writer has finished sending data;
- A Mutex (exclusive lock) guarding writes to *MessageSize* and *ReadCount*;
- Two condition variables: *WriteCond* (notifies readers when a new version is available) and *ReadCond* (notifies the writer when readers have consumed the previous version).

In Figure 5 there may be seen how the memory layout is including a padding that has the size of 40 + C bytes.

4.4.1. Write methods

To perform a NB write, the writer first acquires the Mutex, increments *MessageVersion*, sets *MessageSize*, and copies new data into the *Message* region. It then resets *ReadCount* to zero and signals all readers via *WriteCond* before releasing the Mutex.

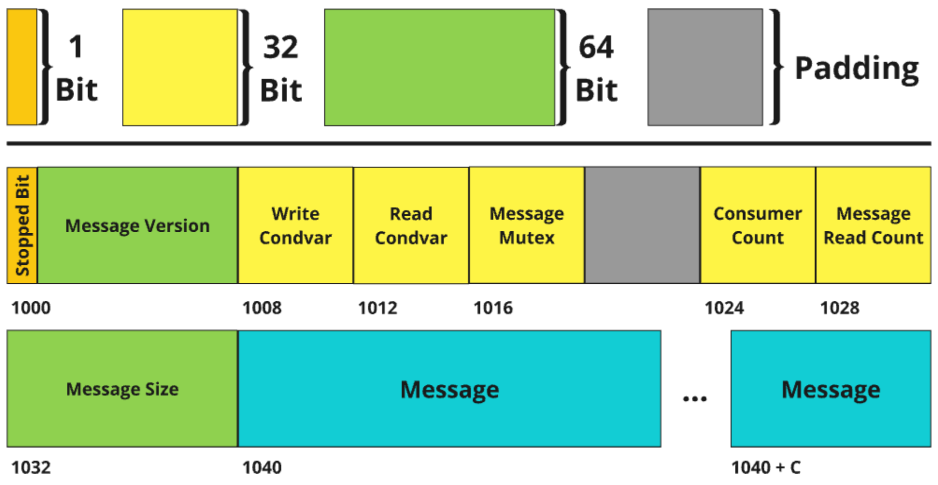

Figure 5. Memory layout of *SharedMessage*: fields are aligned to enable atomic access to *MessageVersion*, while the Mutex and condition variables reside in shared memory

For FB writes, the writer must wait until *Read Count* equals *Consumer Count*, thus ensuring every reader has consumed the previous version. It does so by waiting on *Read Condvar* (releasing the mutex while blocked), then repeats the write steps above. *Partial-blocking* writes only require *Read Count* to reach at least one, allowing the writer to proceed once any reader has consumed the prior version. In both blocking modes, the writer sleeps on *Read Condvar* rather than busy-waiting.

4.4.2. Read methods

A non-blocking reader first reads *MessageVersion* (atomic load). If this version differs from its local copy, the reader acquires the Mutex, re-checks *MessageVersion* (in case it changed while waiting), copies the *Message* into a private buffer, updates its local version, increments *ReadCount*, signals *ReadCond* (in case the writer is blocked), and releases the mutex. If no new version exists, the reader returns immediately.

A blocking reader acquires the Mutex. Then it checks if *MessageVersion* has changed. If not, it waits on *WriteCond* until a new version appears or *Stopped Flag* is set. Once a new version is available (or if *Stopped Flag* indicates no more writes), the reader copies the *Message*, updates *ReadCount* and *MessageVersion*, signals *ReadCond*, and releases the mutex. If *StoppedBit* is set and no unread version remains, the reader returns with no data.

By separating version checks (atomic, without locking) from protected updates (under a mutex), readers encounter minimal overhead checking for new data, and writers avoid unnecessary wakeups. This approach ensures safe, low-latency transfer of large *PipeData* objects among multiple processes.

4.5. Implementation and benchmarking

Three approaches to implementing the *SharedMessage* specification, plus a baseline using Python's standard multiprocessing pipes (MP-Pipe) were evaluated. All implementations were benchmarked on an Intel i5-12600K processor over 50,000 blocking write-read iterations, using mock *PipeData* objects that emulate real data (initial frame, processed frame, and extracted features). Table 1 reports average transfer latencies, while Table 2 shows the total elapsed times for resolutions from 256×256 (576 KB) up to 1920×1080 (17.8 MB).

Table 1. Write-Read Average Latency (ms) 50.000 iterations

Resolution	Size	MP-Pipe	RS-IPC	SHM-LOCK	SHM-PTH
256×256	576 KB	0.247	0.2	0.226	0.233
512×512	2.25 MB	0.908	0.728	0.796	0.82
640×480	2.64 MB	1.019	0.898	0.982	0.994
1280×720	7.91 MB	5.542	4.529	4.762	4.833
1920×1080	17.80 MB	16.161	11.26	11.35	11.96

Table 2. Write-Read Total Transfer Time (s) 50.000 iterations

Resolution	Size	MP Pipe	RS-IPC	SHM-LOCK	SHM-PTH
256×256	576 KB	12.336	10.01	11.306	11.67
512×512	2.25 MB	45.411	36.408	39.789	40.986
640×480	2.64 MB	50.931	44.895	49.12	49.986
1280×720	7.91 MB	277.103	226.434	238.106	241.671
1920×1080	17.80 MB	808.053	562.98	567.252	598.005

SHM-LOCK. The first prototype used Python's multiprocessing module to allocate shared memory and mp.Lock for synchronization. However, *mp.Lock* cannot be stored directly in the shared memory. Instead, it must be passed from parent to child processes, complicating the initialization. Furthermore, Python's condition variables are implemented as locks, so a state change wakes all waiters, even if only some processes need to proceed. Finally, Python offers no native atomic primitives, forcing the operating system to hold the lock whenever checking or updating version and stopped flags - adding overhead and reducing the parallel efficiency.

SHM-PTH. To address the above limitations, the pthread API via Python's ctypes interface was used. By loading and invoking *pthread_mutex* and *pthread_cond* structures directly in shared memory, true in-memory locks and condition variables were obtained. This design aligns with the intended layout and avoids the *mp.Lock* constraints. Nevertheless, it still lacks atomic operations for version or stopped-flag checks, so readers and writers must acquire the mutex even when simply testing for new data, generating an unnecessary latency.

RS-IPC. To eliminate the overhead of Python's missing atomic operations, *SharedMessage* was reimplemented in Rust. Rust's standard library provides atomic types (e.g., AtomicU64) and allows locks and condition variables in shared memory without the indirection required by Python. The Rust functionality to Python was exposed via PyO3 bindings. Although PyO3 adds a small invocation overhead, Rust's ability to release Python's GIL during shared-memory operations enables true multithreading for concurrent readers and writers. As a result, RS-IPC achieves lower latency under contention and can support future optimizations - such as background feeder threads - to push write/read times close to zero in the main thread.

The similarity between RS-IPC (Open Source implementation available at: https://github.com/razvanfilea/rs_ipc), SHM-LOCK, and SHM-PTH in raw transfer times indicates that most latency arises from bulk memory copies, not the synchronization overhead. However, RS-IPC's ability to release the GIL and leverage true multithreading yields greater end-to-end concurrency, especially when multiple frames or feature sets contend for the shared buffer. In future work, Rust-based feeder threads are planned to be implemented to perform asynchronous writes and reads, effectively reducing the main-thread write/read overhead to near zero.

The presented results confirm overall that the shared-memory IPC outperforms the pipe based solutions by a substantial margin, and that a Rust-backed implementation (RS-IPC) offers additional concurrency benefits compared to the pure-Python variants.

5. Evaluation

Benchmark experiments on identical video streams and hardware configurations were conducted in order to evaluate the proposed architecture, the *SharedMessage* specification and the PB (partial-blocking) communication model.

5.1. Proposed solution validation

First, the naive multiprocessing design was measured - one process per branch, sending serialized *PipeData* via pipes, against the proposed shared-memory architecture on a 720p prerecorded video of 2,298 frames at uncapped speed. Each run was repeated 10 times with insignificant variance in results.

Figure 6 shows the real time updating hierarchical timing chart that monitors the entire system for potential bottlenecks.

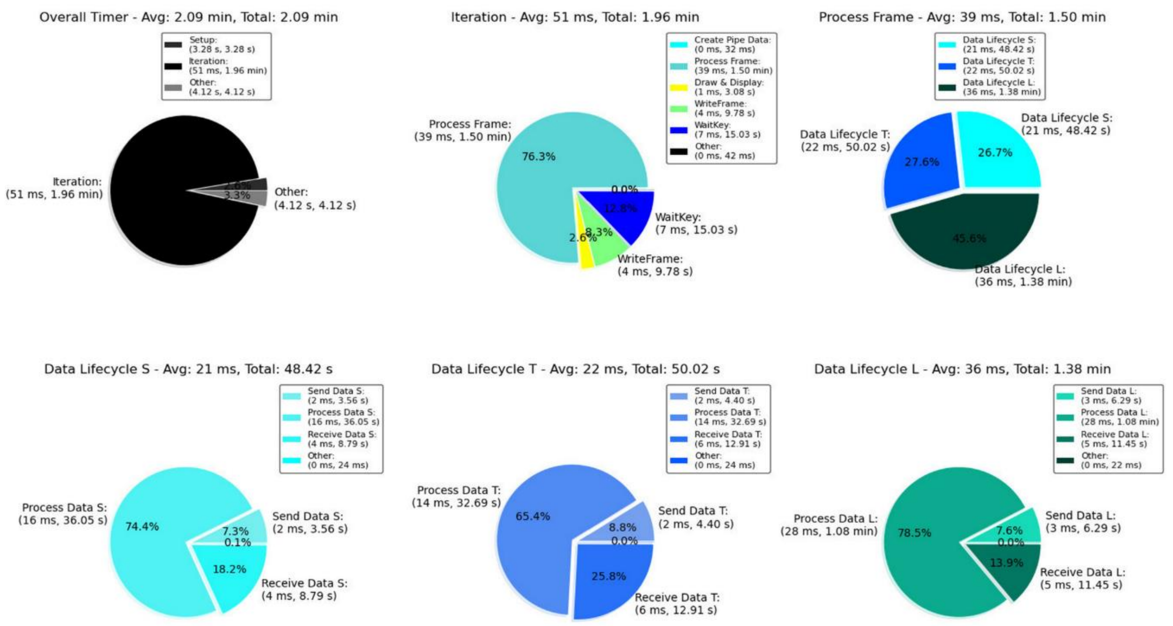

Figure 6. Timing Data for the Naive Multiprocessing Architecture

Since the manager synchronizes all four branches before merging, the system throughput is constrained by the slowest component-Lane Detection (highlighted in teal), which operates at approximately 36 ms per frame. Each processing iteration required 51 ms (corresponding to 20 fps), with 76% of the total time spent awaiting completion of the slowest pipeline and the remaining 24% allocated to I/O and visualization tasks.

The *PipeData* object lifecycle (including instantiation, filter execution, serialization and IPC write/read operations) analysis showed that the filter execution alone covered 60-80% of the iteration duration. The 20-40% was attributed to the IPC overhead.

5.1.1. SharedMessage Partial-Blocking mode

For this mode, each PipeData was instrumented with a Timing object that logs timestamps at each stage: (1) instantiation, (2) per-pipeline processing, (3) transfer to the MPManager, (4) merging, and (5) delivery to decision-making/visualization. The results are shown in Figure 7.

Due to an extra IPC transfer (three processes instead of two), the average transfer latency rose to 20 ms, although the overall system throughput improved significantly due to its ability to use the partial-blocking strategy. In total, processing all frames took 32.39 s (70.94 fps), a 3.5× improvement over the naive design that finished in 114.4 s (20.1 fps).

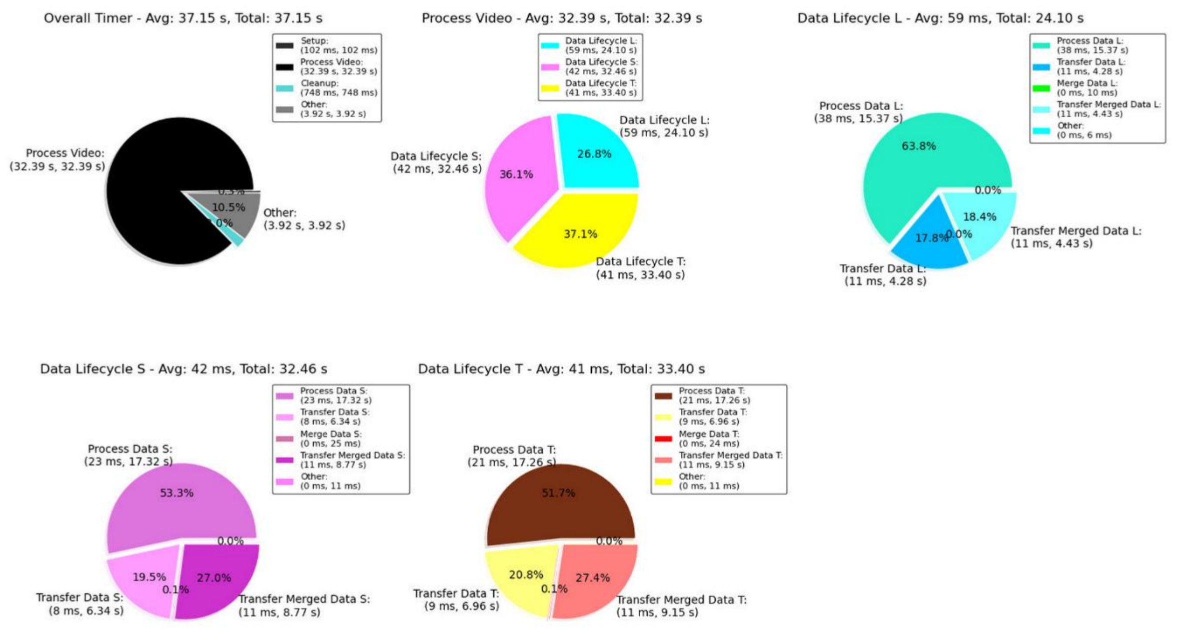

Figure 7. Timing data for the *SharedMessage* architecture

5.2. Hardware-Software benchmarking

Three platforms were evaluated with varying compute and memory resources. Table 3 summarizes their specifications.

Table 3. Hardware Configurations for System Testing

Specification	Jetson AGX Orin	CUDA Workstation	Non-CUDA Workstation
Processor	ARM Cortex-A78AE	Intel i5-12600K	AMD Ryzen 7 7700
Clock Speed	2.2 GHz	3.7 GHz	3.8 GHz
Cores-Threads	12C - 12T	10C - 16T	8C - 16T
RAM	32 GB LPDDR5	32 GB DDR4	32 GB DDR5
GPU	Nvidia Ampere	Nvidia RTX 3090	AMD RX 7700 XT
GPU Mem	32 GB LPDDR5	24 GB GDDR6X	12 GB GDDR6
CUDA Cores	2,048	10,496	N/A

5.3. Operational modes

To evaluate the proposal, three situations that differ in the trade-offs between throughput, visualization, and consistency were defined. Table 4 lists each mode's processing strategy, active pipelines, and visualization setting. They are defined as follows:

- Real-Time (RT) Mode: partial-blocking, minimal visualization (no intermediate frames), representing a live autonomous driving scenario.

- Development (D) Mode: partial-blocking, intermediate visualization enabled for real time monitoring.
- Analysis (A) Mode: full-blocking, intermediate visualization enabled, every frame processed by each pipeline for consistency testing.

The latter two modes operate with the exact same CPU-only pipelines, 5 lane detectors in parallel, to measure the impact of the partial-blocking strategy compared to the full-blocking mode.

Real-Time Mode. In Real-Time Mode, a single lane detection pipeline was run in parallel with two object detection pipelines to simulate a live autonomous driving scenario requiring both CPU and GPU processing. The results in Table 5. illustrate the real-time capabilities of this architecture, maintaining at least 35 fps across all configurations and resolutions, which is more than sufficient for most vision tasks in autonomous driving.

Table 4. Operational Modes and Their Configuration

Mode	Processing Strategy	Active Pipelines	Intermediate Visualization
Real-Time Mode	Partial Blocking	1 Lane Det + 2 Obj Det	Disabled
Dev Mode	Partial Blocking	5 Lane Det	Enabled
Analysis Mode	Full Blocking	5 Lane Det	Enabled

Dev Mode (Partial Blocking) vs. Analysis Mode (Full Blocking). Across all platforms and resolutions, Dev Mode achieves roughly a 4 times speedup over Analysis Mode, as shown in Table 5. These results demonstrate the effectiveness of the proposed partial-blocking strategy, which allows the faster pipelines to progress without waiting for the slower ones, while still fusing results from all branches as they become available. Enabling the full-blocking results in similar performance to the naive architecture in Figure 3, with its implicit full-blocking, which is in accordance with the theoretical analysis. A demo of Partial vs Full Blocking on Real-Time Scenario with Intermediate Visualizations can be seen at: https://www.youtube.com/watch?v=HKS9mA1-57E.

Table 5. Dev Mode vs. Analysis Mode Total Times with FPS and Speedup

Resolution	Hardware	Dev Mode (PB)	Analysis Mode (FB)	Speedup
640×480	CUDA Workstation	24.67 s (93.1 fps)	114.60 s (20.1 fps)	4.63×
640×480	Non-CUDA Workstation	16.40 s (140.0 fps)	71.40 s (32.2 fps)	4.35×
640×480	Jetson AGX Orin	37.50 s (61.3 fps)	148.00 s (15.5 fps)	3.95×
1280×720	CUDA Workstation	34.63 s (66.3 fps)	152.40 s (15.1 fps)	4.39×
1280×720	Non-CUDA Workstation	31.40 s (73.3 fps)	120.00 s (19.2 fps)	3.82×
1280×720	Jetson AGX Orin	60.63 s (37.9 fps)	226.63 s (10.1 fps)	3.74×
1920×1080	CUDA Workstation	59.16 s (38.8 fps)	275.40 s (8.3 fps)	4.67×
1920×1080	Non-CUDA Workstation	53.01 s (43.4 fps)	205.80 s (11.2 fps)	3.88×
1920×1080	Jetson AGX Orin	122.23 s (18.8 fps)	453.03 s (5.1 fps)	3.71×

5.4. Real-World application

The final experiment aimed to validate the usefulness of the proposed architecture in a real scenario. The vehicle depicted in Figure 8 was designed for an international 1:10 scale autonomous driving competition in which the authors participated (Kilyen et al., 2021). The Hardware configuration is based on a Raspberry Pi 5 with a 720p camera and a Nvidia Jetson AGX Orin for CUDA accelerated object detection.

The control stack was developed and validated in a simulation environment (Szilagyi, Benta & Sacarea, 2024)) and then integrated with the perception pipeline from Figure 2 using the proposed architecture. (demo is available at: https://www.youtube.com/watch?v=pzqnW4tQQjw).

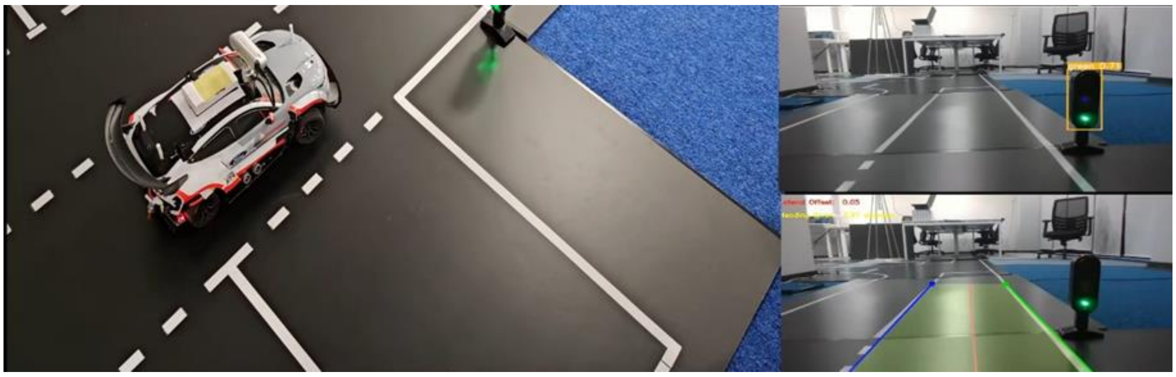

Figure 8. RC vehicle performing traffic light detection, path detection, trajectory planning and following on a test track

5.5. Summary

The results of the experiments show that the naïve multiprocessing design is indeed limited by the slowest branch and suffers from high IPC overhead (51ms per frame, 20fps on the CUDA Workstation), while the shared-memory version achieves 70.94 fps on the same hardware, which is around a 4 times improvement even with extra inter process transfers. Using the authors' *SharedMessage* Architecture in partial-blocking communication mode allows for real-time performance even on embedded devices like the Jetson AGX Orin (35 fps) and can be used for real-world applications.

6. Limitations and future work

While the proposed architecture has demonstrated promising results, several limitations and areas for potential improvement remain:

Asynchronous *SharedMessage* Write and Read Operations: The Shared Memory I/O operations are currently synchronous and could be further optimized. Thus, a key opportunity is to leverage Rust's multithreading capabilities to implement background feeder threads for asynchronous write and read operations, and so to further reduce the execution time in the calling process.

FreeThreaded Python: With the release of Python 3.13 Freethreaded may offer improved concurrency by mitigating some of the Global Interpreter Lock (GIL) constraints. This could unlock significant performance gains in multi-threaded contexts. The authors plan to develop a prototype version of this architecture in Python 3.13 and compare its performance against the existing implementation.

JSON-Based Pipeline Configuration and Editing: Currently, any modification to the pipeline involves direct code modifications, which are cumbersome, especially if one does not have a deep understanding of the underlying architecture. A JSON-based configuration system paired with a user-friendly interface would greatly simplify the development by allowing users to add, remove, reorder, change parameters of filters and more dynamically without altering the codebase.

Benchmark Differing Autonomous Driving Perception Pipelines: While the proposed architecture is designed to be flexible and adaptable to various perception tasks, currently tested pipelines were developed for the Bosch Future Mobility Challenge (Kilyen, 2021) and offer a limited selection of components that should be expanded. Future work should focus on integrating and benchmarking the additional pipelines from the literature to validate the architecture in more contexts.

7. Conclusions

This paper introduces a modular, concurrent vision architecture that balances performance with developer accessibility in an effective manner, useful for robotics applications. Due to the combination of Python's rich ecosystem of computer vision libraries and Rust's low-level efficiency and safe concurrency, this proposed architecture addresses the common bottlenecks, in particular the IPC overhead and the limitations imposed by Python's GIL. The RT performance has been achieved through a shared memory design that reduces unnecessary data copies and enables the partial-blocking communication mode. It also allows the faster pipelines to operate without being stalled by the slower ones.

The experimental results on multiple hardware platforms (including embedded devices and high-end workstations) demonstrate its superior scalability, resource efficiency, and a throughput that is compared to the naive multiprocessing or parallel designs based on the standard Python IPC primitives.

The real-world like testing on a 1:10 scale remote control car further validates the practicality of this approach by showing its adaptability and responsiveness in an actual autonomous driving setting. The architecture's flexibility was further highlighted by supporting both RT and offline development modes, permitting convenient debugging, visualization, and data recording.

While the proposed system shows promising results, additional optimizations (such as asynchronous read/write in Rust and leveraging Python's future free-threading capabilities) present opportunities for even greater performance gains in the next works.

This approach offers a robust foundation for executing vision pipelines in RT, where timeliness and efficiency are at the very top.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al. (2016) *TensorFlow: Large-scale machine learning on heterogeneous distributed systems*. [Preprint] https://arxiv.org/abs/1603.04467 [Accessed: 28th July 2025].

Bradski, G. (2000) The OpenCV library. *Dr. Dobb's Journal of Software Tools.*

Flitton, M. (2022) Speed Up Your Python with Rust: Optimize Python Performance by Creating Python Pip Modules in Rust with PyO3. Birmingham, UK, Packt Publishing.

Gu, Y., Wang, Q. & Qin, X. (2021) Real-time streaming perception system for autonomous driving. In: *2021 China Automation Congress (CAC).* IEEE. pp. 5239–5244.

Jahromi, B.S., Tulabandhula, T. & Cetin, S. (2019) Real-time hybrid multi-sensor fusion framework for perception in autonomous vehicles. *Sensors.* 19(20), 4357. doi:10.3390/s19204357.

Kam, H.R., Lee, S.H., Park, T. & Kim, C.H. (2015) RViz: A toolkit for real domain data visualization. Telecommunication Systems. 60, 337–345. https://doi.org/10.1007/s11235-015-0034-5.

Kilyen, N.A., Lemnariu, R.F., Mois, G.D., Chen, Y., Morris, B.T. & Muntean, I. (2021) The IEEE ITSS and Bosch Future Mobility Challenge: A hands-on start to autonomous driving [technical activities]. *IEEE Intelligent Transportation Systems Magazine*. 13(3), 276–282. doi:10.1109/MITS.2021.3081939.

Macenski, S., Foote, T., Gerkey, B., Lalancette, C. & Woodall, W. (2022) Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics*. 7(66), eabm6074. doi: 10.1126/scirobotics.abm6074.

Meunier, R. (1995) The pipes and filters architecture. In: Coplien, J.O. & Schmidt, D.C. (eds.) *Pattern Languages of Program Design*. Reading, MA, Addison-Wesley. pp. 427–440.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019) PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*. 32, 8024–8035. https://papers.nips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html [Accessed: 28th July 2025].

Ros, G., Ramos, S., Granados, M., Bakhtiary, A., Vázquez, D. & Lopez, A.M. (2015) Vision-based offline-online perception paradigm for autonomous driving. In: *2015 IEEE Winter Conference on Applications of Computer Vision.* IEEE. pp. 231–238. doi:10.1109/WACV.2015.38.

Rosique, F., Navarro, P.J., Fernandez, C. & Padilla, A. (2019) A systematic review of perception system and simulators for autonomous vehicles research. *Sensors.* 19(3), 648. doi:10.3390/s19030648.

Scott, M.L. & Brown, T. (2013) *Shared-Memory Synchronization*. New York, Springer.

Sridharan, M. & Stone, P. (2005) Real-time vision on a mobile robot platform. In: *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. pp. 2148–2153. doi: 10.1109/ IROS.2005.1545540.

Szilagyi, D., Benta, K.I. & Sacarea, C. (2024) Benchmarking autonomous driving systems using a modular and scalable simulation environment. In: *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI).* IEEE. pp. 420–426. doi:10.1109/ ICTAI62512.2024.00067.

Xu, Y., Zhang, F., Sun, Y, (2025) Intelligent Assisted Driving Method Based on Mathematical Modeling and MPC Algorithm. *Studies in Informatics and Control*. 34(2), 15-25. doi:10.24846/v34i2y202502.

Dávid SZILÁGYI is a Master's student at the Faculty of Mathematics and Computer Science, Babes-Bolyai University, Cluj-Napoca, Romania. He is pursuing a master's degree in has a strong background in computer science, machine learning and robotics. His current research interests include AI-driven simulation environments, high-performance perception and planning modules, and system architectures for scalable robotic systems. He has contributed to multiple European research Computer Science with a focus on autonomous driving, robotics, and simulation technologies. He projects and is passionate about developing open, accessible technologies that foster interdisciplinary innovation.

Dávid SZILÁGYI este student la masterat în cadrul Facultății de Matematică și Informatică a Universității Babeș-Bolyai din Cluj-Napoca, România, urmând un program de master în informatică, cu accent pe conducerea autonomă, robotică și tehnologii conexe. Are o pregătire solidă în Informatică, învățare automată și robotică. Interesele sale de cercetare includ medii de simulare bazate pe inteligență artificială, module de percepție și planificare de înaltă performanță și arhitecturi de sistem scalabile pentru aplicații robotice. A contribuit la multiple proiecte europene studențești și este pasionat de dezvoltarea de tehnologii deschise și accesibile, care să favorizeze inovarea interdisciplinară.

Răzvan FILEA is a Master's student at the Faculty of Mathematics and Computer Science, Babes-Bolyai University, Cluj-Napoca, Romania. He is pursuing a master's degree in Computer Science with a focus on embedded systems and distributed computing, and has a strong background in C++ and Rust. His research interests include real-time capable architectures, edge computing for robotics (processing data locally on devices to reduce latency and improve responsiveness) and high frequency communication protocols for distributed systems. He has contributed to several research projects and is passionate about developing reliable low-level infrastructure for next-generation autonomous and robotic platforms.

Răzvan FILEA este student la masterat la Facultatea de Matematică și Informatică, Universitatea Babeș-Bolyai, Cluj-Napoca, România. Urmează un program de masterat în Informatică, cu specializare în sisteme încorporate și calcul distribuit, și are o vastă experiență în C++ și Rust. Domeniile sale de interes în cercetare includ arhitecturi capabile să funcționeze în timp real, calcul la periferie pentru robotică (procesarea datelor la nivel local pe dispozitive pentru a reduce latența și a îmbunătăți capacitatea de răspuns) și protocoale de comunicații de înaltă frecvență pentru sisteme distribuite. A contribuit la mai multe proiecte de cercetare și este pasionat de dezvoltarea unei infrastructuri fiabile de nivel inferior pentru platforme autonome și robotice de nouă generație.

Kuderna-Iulian BENȚA received his Ph.D. in Electronics and Telecommunication Engineering in 2011. He is currently a tenured lecturer et Babes-Bolyai University, Romania, Faculty of Mathematics and Computer Science, Department of Computer Science. To date, he has published 8 books and over 30 articles in scientific journals and conference proceedings. Since

2004, he has participated as a member of research teams in over 13 national and international projects. His research interests include Affective Computing, Pervasive Computing, and Human-Computer Interaction.

Kuderna-Iulian BENȚA a obținut titlul de doctor în Inginerie electronică și telecomunicații în 2011. În prezent, este lector titular la Universitatea Babeș-Bolyai din România, Facultatea de Matematică și Informatică, Departamentul de Informatică. Până în prezent, a publicat 8 cărți și peste 30 de articole în reviste științifice și lucrări în conferințe. Din 2001, a participat ca membru al echipelor de cercetare la peste 13 proiecte naționale și internaționale. Domeniile sale de interes în cercetare includ Calculul afectiv, Calculul omniprezent și Interacțiunea om-calculator.