

# Demonstrating the Advantages of Computational Offloading of XR Services via WebAssembly

Gabriel Espindola\*, Matheus Pires\*, Gustavo Spadotto†,

Cristiano Both†, Bruno Silvestre\*, Kleber Cardoso\*, Andrew Williams‡, Fabio Verdi||, Sand Correa\*

\*Universidade Federal de Goiás (UFG), Brazil, †Sinus River Valley University (UNISINOS), Brazil

‡Ericsson Research, Stockholm, Sweden, || Universidade Federal de São Carlos (UFSCar), Brazil

{gabriel.nery, matheuspires23}@discente.ufg.br\*, {sandluz, kleber, brunoos}@ufg.br\*,

{gustavo, cbboth}@unisinis.br†, andrew.williams@ericsson.com‡, verdi@ufscar.br||

**Abstract**—Extended reality (XR) services form the basis of various innovative applications. These new applications are expected to run on mobile devices with limited computational and energy capabilities. At the same time, XR services should fulfill some expectations regarding data rates and end-to-end latency to guarantee an uninterrupted user experience. In this context, offloading intensive computation to the edge is particularly advantageous for mobile devices, enabling access to more capable processing hardware. Current studies on computational offloading focus mainly on the compute and network continuum formed between edge and cloud and realized mostly through containers. Conversely, this demonstration showcases a computational offloading framework that allows the expansion of XR services functionality from the connected mobile device to an edge computing environment. The presented framework is based on a portable, lightweight, and secure WebAssembly runtime and uses open technology implementations. We demonstrate that the developed framework allows significant performance improvements in a Yolo object detection application and reduces heat generation on mobile devices.

## I. INTRODUCTION

The next generation of immersive multimedia and connectivity services is called extended reality (XR) and includes virtual, augmented, and mixed-reality technologies [1]. XR services include functionalities such as simultaneous localization and mapping (SLAM), pose estimation, object detection, hand tracking, and semantic segmentation, which have been used in various innovative applications, from the industrial metaverse to entertainment and healthcare. On the one hand, various of these new applications are expected to run on mobile user equipment (UE) such as smartphones, XR headsets, or drones, which usually have limited computational capabilities and energy capacity. On the other hand, to guarantee an uninterrupted user experience, XR services should fulfill some expectations regarding data rates and end-to-end latency [1]. A promising way to overcome this problem is to offload the XR services of a running application from the UE to the network edge, where a server can increase the application’s quality of experience by giving offloaded tasks remote access to, e.g., hardware accelerators such as GPUs.

Computational offloading has been approached in the compute and network continuum formed between edge and cloud. In such context, ETSI-MANO-aligned implementations and Kubernetes distributions are employed to move application

functionality running in the cloud to edge [2]. However, such platforms are inadequate when moving running XR services from the UE to the edge. Virtual machines have a large footprint and high overhead, while containers are generally not portable across operating systems. To deal with the expectations regarding data rates and end-to-end latency of XR services and the heterogeneity of UEs, the dynamic computational offloading framework should be platform-independent (**R1**), lightweight (**R2**), and have a low footprint (**R3**). This framework must also be secure (**R4**), providing tenant isolation. Finally, it is also desirable to be language-independent (**R5**) to allow developers more flexibility.

In [3], a distributed execution framework and a novel programming model for computational offloading based on WebAssembly (Wasm) runtimes was introduced. Wasm is an instruction format designed to be executed on load-time efficient, memory-safe, and sandboxed stack-based virtual machines [4]. The work proved the solution’s viability for building an offloading framework that satisfies requirements **R1-R5**, showing performance and power consumption improvements through computational offloading in resource-constrained devices. However, the toolchain used in [3] is optimized for Swift applications and relies on proprietary system interfaces to work efficiently. In addition, this toolchain has proven challenging to develop and extend.

This demonstration seeks to fill this gap by showcasing a feature equivalent offloading framework developed in [3] but using open technologies more suited to WebAssembly implementations, such as Rust and C++. We then evaluate the implemented framework’s effectiveness in dealing with XR requirements, showing the application’s performance with regard to throughput, CPU usage, and CPU temperature. Through this demonstration, we contribute to the design of offloading frameworks for scenarios requiring portable and secure runtimes. Since Wasm is still an immature technology compared to more conventional portable bytecode formats such as Java or to more heavyweight distributed platforms such as containers or virtual machines, demonstrations like the one presented in this paper are crucial to making sure that future Wasm-related standards meet the requirements of advanced XR use-cases.

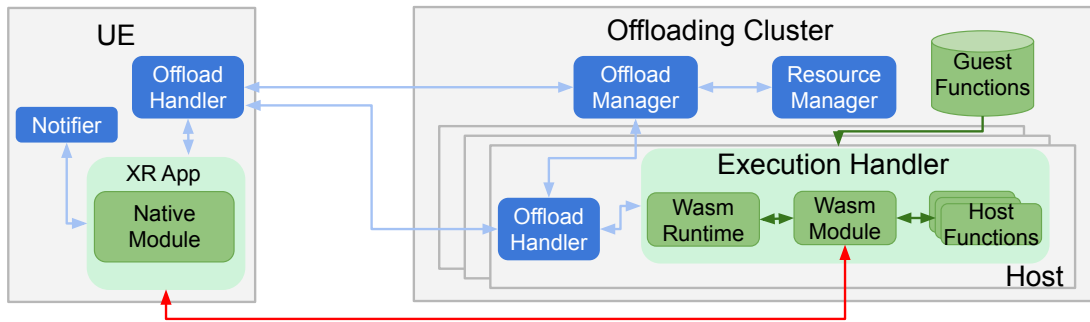


Fig. 1. The dynamic computational offloading architecture. Boxes and arrows in blue represent control panel elements; boxes and arrows in green represent execution plane elements, while arrow in red represents the data plane.

## II. SYSTEM ARCHITECTURE

We devise a framework architecture where offloading is handled at the granularity of a module. The latter is a part of the application that can be offloaded, including single functions, tasks, or the whole application. Figure 1 depicts the overall architecture, which comprises three planes.

The control plane consists of functionalities to manage offload requests and allocate edge resources. It is divided into two parts: the UE side and the edge offloading cluster side. On the UE side, the control plane comprises (i) a Notifier, which gathers device metrics, and (ii) a (UE) Offload Handler, responsible for handling the interaction with the application, discovering the closest offloading cluster, and requesting the offload of a specified module with its corresponding part on an allocated host within the offloading cluster. On the offloading cluster side, the control plane comprises (i) an Offload Manager that coordinates activities such as authentication, authorization, and resource management for each offloading request coming from the UE side; (ii) a Resource Manager, responsible for selecting a host to run the module; and (iii) a (Host) Offload Handler, responsible for synchronizing an offloading event with its counterpart in the UE.

The execution plane consists of the functionalities to load and execute a specified module. It comprises (i) the running application on the UE and (ii) the Execution Handler running on the selected host within the offloading cluster. The Execution Handler acts as the remote application, and it is responsible for running the offloaded module. Since the execution of offloaded modules is based on Wasm, the Execution Handler embeds a WebAssembly runtime to load Wasm modules and call exported functions, known as *guest functions*. The Wasm modules (and the guest functions) can be obtained from a repository within the edge offloading cluster. Following the WebAssembly programming model, the Wasm module can access host machine functionality through the standardized WASI system interface<sup>1</sup> or non-standard *host function* calls to host resources (e.g., networking, GPU). The Execution Handler interacts with the (Host) Offload Handler, allowing for the invocation of Wasm modules and the calling of exposed

Wasm functions. Once instantiated, the offloaded module communicates directly with the UE application through the data plane rather than the service framework.

To guide the developer in the task of dividing the application functionality, the framework provides a communication method known as *elastic channel*, where endpoints can transparently change location during their lifetime. This allows one endpoint to remain fixed in the UE while the other endpoint can freely move around, e.g., to the closest offloading cluster. Elastic channels are included in the application as a library, allowing endpoints to be explicitly declared by the developer. Complementing the elastic channel programming model, a form of Remote Procedure Call, known as *elastic functions*, allows a function call to look the same both when a module containing the function implementation is residing locally in the UE or when it is at a remote location. To do that, an extra pre-processing step in the compiler permits the developer to tag functions as offloadable. When compiled, two versions of an offloadable function are built, one that can run natively on the device and another compiled to WebAssembly that can readily run remotely, independent of the platform choices at the offloading cluster.

## III. DEMO SETUP AND RESULTS

Since our objective is to investigate the viability of using open technologies to build an offloading framework based on the WebAssembly programming model, we limit the scope of the demonstration to the execution plane. Figure 2 describes the proof-of-concept (PoC) deployment. We use two machines with different computing capabilities. The first is a desktop comprising an i7-3770 3.9 GHz processor with 8GB RAM. Since we are not focused on the control and data plane signaling in this demonstration, the desktop is used as the less capable processing machine, taking the place of the UE. The second machine represents the edge host and comprises an Intel Xeon Gold 5418Y 3.800 GHz processor, 128GB RAM, and an NVIDIA RTX A5000 GPU. The two machines are connected by a LAN. The XR application uses the YoloV8 neural network for object detection based on OpenCV 4.10. The application reads a video stream, decoding frame by frame (codification function). Each frame is then

<sup>1</sup><https://github.com/WebAssembly/WASI/>

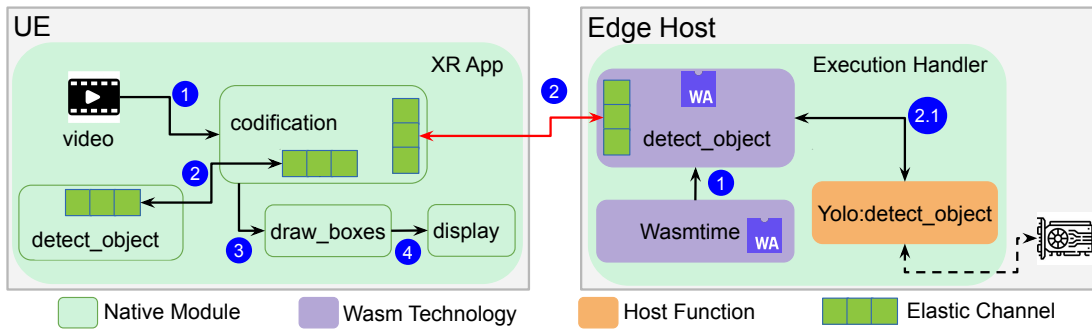


Fig. 2. PoC implementation and deploy.

converted to an image and processed by the YOLO model, which detects bounding boxes, object classes, and confidence scores (detect\_object function). Bounding boxes and object labels are drawn on the frame (draw\_boxes function), and the enriched frame is displayed to the user (display function). The detect\_object function is tagged as offloadable; thus, it is compiled to native and Wasm code. The application and the Execution Handler are developed in Rust, while OpenCV is implemented in C++ and compiled to enable GPU execution. We use the Wasmtime 26.0.1 WebAssembly runtime and elastic channels use the TCP protocol.

functionality to the edge host. The edge host processes the next 50 frames when the application switches back the execution of the detection functionality to the UE. This process is repeated until the end of the video. Figure 3 shows FPS, UE CPU usage, UE CPU temperature, and network data results. We can observe that when detect\_object is running locally (frames 0-50, 100-150, 200-250, 300-350, 400-450, and 500-550), the frame rate is around 7 FPS, UE CPU utilization is approximately 60%, UE CPU temperature grows up to 57.5° celsius and no data is transmitted in the network. When detect\_object is offloaded to the edge host, the frame rate grows to 20 FPS due to the use of GPU, UE CPU usage and temperature decreases, and network data reaches more than 1.2 Mb. These results show that by using open technologies, we can implement an easy to use/extend WebAssembly execution environment while increasing the application’s performance and reducing energy heat on the UE.

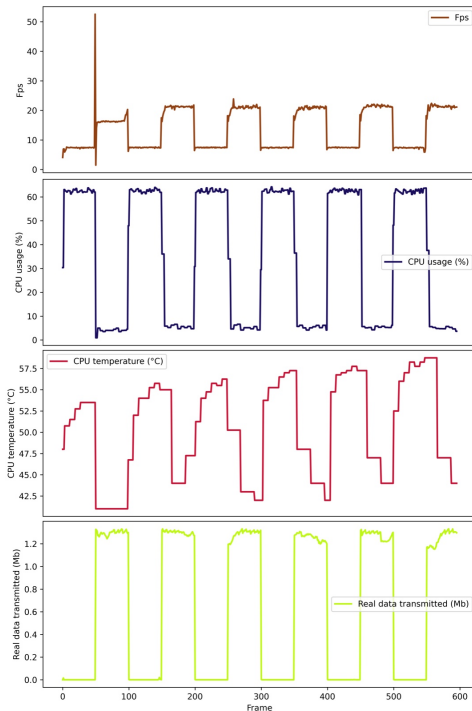





Fig. 3. FPS, CPU, temperature, and data network measurements.

To make evaluating the impact of offloading simple, we design the application to switch between offload and non-offload mode at each 50 frames. Initially, all the application components run natively on the UE. After the first 50 frames are processed, the application offloads the object detection

#### IV. CONCLUSIONS

This demonstration contributed to the research and development efforts to realize a lightweight, platform-independent, language-agnostic, and easy-to-use computational offloading framework. These efforts will pave the way toward ensuring that future Wasm-related standards meet the requirements of advanced use cases.

#### ACKNOWLEDGMENT

This work was supported by Ericsson Telecomunicações Ltda.,  and by the São Paulo Research Foundation (FAPESP),  grant 2021/00199-8, CPE SMARTNESS .

#### REFERENCES

- [1] A. Amiri *et al.*, “Application Awareness for Extended Reality Services: 5G-Advanced and Beyond,” *IEEE Communications Magazine*, vol. 62, no. 8, pp. 38–44, 2024.
- [2] T. Taleb *et al.*, “Toward Supporting XR Services: Architecture and Enablers,” *IEEE Internet of Things Journal*, vol. 10, no. 4, pp. 3567–3586, 2023.
- [3] V. Yadhav, A. Williams, O. Smid, J. Kjällman, R. Islam, J. Halén, and W. John, “Dynamic Computational Offloading for Mobile Devices,” in *Proceedings of the 14th International Conference on Cloud Computing and Services Science - CLOSER*, ser. CLOSER’24. SciTePress, 2024, pp. 265—276.
- [4] A. Haas *et al.*, “Bringing the web up to speed with WebAssembly,” *SIGPLAN Not.*, vol. 52, no. 6, p. 185–200, 2017.