Towards Liquid AI in IoT with WebAssembly: A Prototype Implementation

Pyry Kotilainen[1], Ville Heikkilä[2], Kari Systä[2], and Tommi Mikkonen[1]

[1]Faculty of Information Technology
University of Jyväskylä, Jyväskylä, Finland

[2]Faculty of Information Technology and Communication Sciences
Tampere University, Tampere, Finland

pyry.kotilainen@jyu.fi, ville.heikkila@tuni.fi, kari.systa@tuni.fi, tommi.j.mikkonen@jyu.fi

Abstract. An Internet of Things (IoT) system typically comprises numerous subsystems and devices, such as sensors, actuators, gateways for internet connectivity, cloud services, end-user applications, and analytics. Currently, these subsystems are built using a wide range of programming technologies and tools, posing challenges in migrating functionality between them. In our previous work, we have proposed so-called liquid software, where different subsystems are developed using a consistent set of technologies and functions can flow from one computer to another. In this paper, we introduce a prototype implementation of liquid artificial intelligence features, which can be flexibly deployed at the cloud-edge continuum.

Keywords: Liquid software · Artificial intelligence · AI · Machine learning · ML · Web of Things · WoT · Internet of Things · IoT · Isomorphic software.

1 Introduction

Contemporary Internet of Things (IoT) systems and their associated applications are capable of generating and managing vast volumes of data. This has paved the way for the utilization of Machine Learning (ML) and Artificial Intelligence (AI) in several application domains, including smart homes, smart cities, healthcare, retail, and industrial systems. However, not all data generated by IoT devices can be transmitted to the cloud for processing due to concerns related to privacy, latency, or limited connectivity. As a result, it becomes imperative to perform certain computations near the data source, while other computations can be offloaded to the cloud. Nevertheless, there are numerous use cases where seamless data and computation transfer between different system components is necessary.

Performing such transfer in the cloud-edge continuum has been one of the goals of so-called liquid software in the IoT domain [16]. A recent literature study defined cloud continuum as "an extension of the traditional cloud towards

multiple entities (e.g., edge, fog, IoT) that provide analysis, processing, storage, and data generation capabilities" [20]. Given the rapidly increasing use of ML technologies, we expect that the same technical challenges that apply to conventional computations shall emerge also in the context of ML technologies across the cloud-edge continuum.

In this paper, we report the first prototype of using AI/ML technologies in the liquid context, building on our previous work [15,16,18]. As the underlying technology framework, we use WebAssembly, a technique for running small memory virtual machines with binary bytecode [23], and neural networks, which are a commonly used technique in ML and has several practical use cases. Moreover, it is expected that new, improved hardware platforms will allow distributing ML functions between the cloud-fog-edge continuum.

The rest of the paper is structured as follows. In Section 2, we introduce the background of the paper. In Section 3, we present our prototype implementation, where we test AI/ML features in liquid fashion. In Section 4, we discuss the key findings. Finally, towards the end of the paper, in Section 5, we draw some conclusions.

2 Background and Motivation

2.1 Isomorphic IoT Systems

Using the term isomorphism – a well-established mathematical concept – in software development has emerged relatively recently. In the context of web applications, isomorphism refers to running the same code in both the backend (cloud) and frontend (web browser) [25].

In general, isomorphic software architectures include software components that do not need to be altered ('change their shape') while running on various hardware or software components of the system. Well-known examples of isomorphic systems are Java and its 'write once, run everywhere' guarantee [2], Unity 3D engine, Universal Windows platform, which allows running the same code on Windows 10, Xbox One gaming machines, and HoloLens devices, and liquid web applications [19].

There are several levels of isomorphism that can be identified [18]. Static, development-level isomorphism allows using the same development technologies consistently throughout the entire system's different computational elements. In contrast, *dynamic* isomorphism allows the usage of a common runtime engine or virtualization solution to enable running the same code on various computational elements without the need for recompilation. In a more sophisticated system, dynamic code migration from one computational element to another is also possible.

In the IoT context, we are interested in dynamic isomorphism, allowing the deployment of the the same, isomorphic software throughout the end-to-end system to run on edge devices, gateways, mobile clients and cloud services, instead of them all running separate software (Figure 1 [27]). Such facilities would liberate the developers from designing dedicated applications for individual nodes in

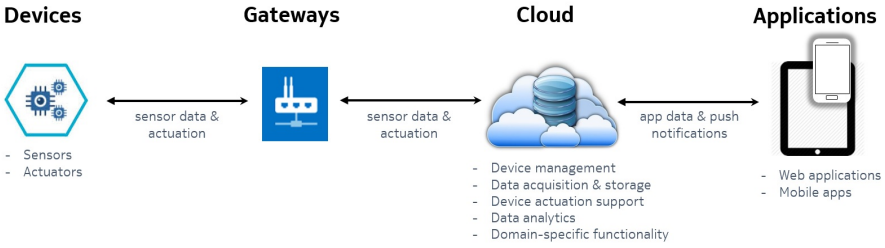

Fig. 1. Elements of a typical IoT end-to-end system, with each element featuring their own implementation technologies.

the network taking into account the associated implementation technologies [28], and only create one implementation that can be deployed to various locations [18].

2.2 Liquid Software

Liquid software [10,11,29,19] is a paradigm that builds on isomorphic software, allowing the migration of software from one computer to another on the fly. In the context of the web, the main use case has been experience roaming from one display to another [19], whereas in the context of IoT, liquid software enables flexible configurations of applications, instead of rigid architectures that are associated with traditional technologies [28].

It has been pointed out that there are various ways to build liquid software, depending on what are the desired characteristics and use cases [7,8]. These characteristics and use cases have an impact on various technical decisions, including topology, replication and migration techniques, thickness of the client, and user interface adaptation, to name some examples.

To support isomorphic, liquid software deployment, a runtime environment is needed where the same infrastructure is made available across the cloud-edge continuum. In our previous work, we have used WebAssembly, with some early results published in [12]. This setup allows using various programming languages to create the software, but the infrastructure maps everything to the WebAssembly virtual machine. This virtual machine can be included in various nodes in the cloud-edge continuum, so that the actual code can be flexibly run in different locations. This added flexibility then enables orchestrating functions so that energy consumption, communication bandwidth, and performance and memory requirements can be taken into account. In fact, similar executions could be run in different configurations at different times, if the circumstances change.

2.3 Liquid AI/ML

One important use case for liquid software is edge intelligence [21]. In "classic" IoT systems, the majority of computation and analytics are performed in the

cloud in a centralized fashion. However, in recent years there has been a noticeable trend in IoT system development to move intelligence closer to the edge, challenging the existing rigid design space [28].

Historically, the computing capacity, memory and storage of edge devices were limited. Due to increasing computational capabilities of edge devices and requirements for lower latencies, though, intelligence in a modern end-to-end computing system is gradually moving towards the edge, first to gateways and then to devices. Another driver is the huge data that the edge devices generate. It is often reasonable to process data on the edge devices for performance reasons, but privacy and data ownership concerns also support placing of the computation to edge devices.

This trend towards edge includes both generic software functions, and – more importantly – time critical AI/ML features for processing data available on the edge with minimal latency. The requirement to run advanced AI/ML and analytics algorithms on the edge increases the demand for consistent programming technologies across the end-to-end system. Hence, in addition to liquid software, we also need liquid AI/ML [26], allowing flexible deployment of intelligent components in different nodes of the IoT network.

2.4 WebAssembly and WASI

WebAssembly (Wasm) is a binary instruction format for a stack-based virtual machine designed for efficiency together with hardware- and platform-independence among other things [22]. WebAssembly offers dynamic isomorphism, as a standard runtime interpreter is used to execute the code. Therefore, WebAssembly can be used as a runtime environment for applications developed using different languages, but compiled for the WebAssembly stack machine.

While the origins of WebAssembly are inside the browser, the developer community has started to realise its significance outside the browser, in particular as a unifying environment for heterogeneous devices [4,34,14,35]. WebAssembly's conservative memory usage and somewhat near-native performance make it a good candidate for constrained environments like IoT devices [9].

Moreover, extensions such as *WebAssembly System Interface* (WASI) [5] have been introduced to access system resources when running WebAssembly outside the browser. This is particularly pertinent to our research, as access to host functions from the WebAssembly runtime offers a way to outsource running AI/ML models to a host runtime with potentially significant performance gains. This approach has been formalized in a WASI proposal called wasi-nn [3], an extension for the WebAssembly System Interface to provide an API to run ML models on a native ML runtime.

The motivation for the wasi-nn proposal are the challenges that would arise from trying to ensure high-performance AI/ML inference in WebAssembly – AI/ML requires special hardware support (GPU, TPU, and special CPU instructions in particular) for maximal performance. Support for these would be challenging to implement for WebAssembly. In addition machine learning still evolves rapidly, with new computational operations getting introduced. These

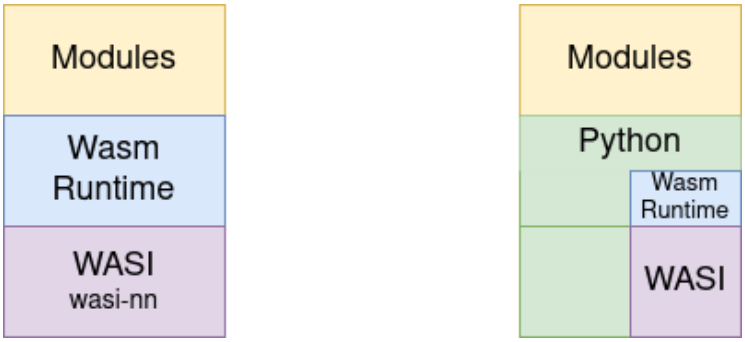

Fig. 2. Two WebAssembly use cases: one where WASI extended with wasi-nn is sufficient to run the module, and other where Python is used to embed WebAssembly runtime and provide more interfacing options with the host than WASI

new operations would need added support to run new models which are using these operations. The wasi-nn proposal therefore leaves the ML runtime implementation details outside WebAssembly domain. This approach has the benefit of protecting the model intellectual property, as the WebAssembly module and runtime do not need to know anything about the inner workings of the model. Essentially wasi-nn treats AI/ML model as a virtualized I/O type, with defined operations such as load and compute.

The proposal has initial focus on only supporting inference, as it is the main AI/ML use case, not training. However, adding support for training has not been excluded from forthcoming implementations. The API provided by wasi-nn is a simple model loader API, inspired by WebNN's model loader proposal [33]. The proposal is framework and model format agnostic, but runtime implementations need to make decisions about what ML runtimes and model formats to support.

The first experimental implementation of wasi-nn for the Wasmtime WebAssembly runtime only supports OpenVINO ML runtime [30]. The implementation is however not considered production quality and Wasmtime needs to be compiled with wasi-nn support enabled to have access to it. However, at present the WasmEdge [6] runtime seems to have the most extensive implementation for wasi-nn, with support for OpenVINO, PyTorch [31] and TensorFlow-Lite [32] ML runtimes.

2.5 Device Management and Orchestration

IoT systems are often composed of many headless devices that need to be remotely managed, and thus most commercial IoT platform have a device management functionality. This functionality keeps track of the devices and the software in them. From the application point of view the device management should

– keep track of devices that can receive applications

- maintain and utilize knowledge about device differences in a heterogeneous fleet of devices,
- control the life-cycle of the applications, including installation, update and removal, and
- in the context of this research manage the (liquid) lifecycle of the devices.

For liquid software this management functionality becomes more challenging since the computation may change its location dynamically. The management functionality should support dynamic moving of running applications – including ML models.

The piece of software or ML model executing on a device is typically a component of a bigger system, and the components need to collaborate for the overall goal. Ensuring this collaboration is called *orchestration* or *choreography* depending on the selected technical approach.

In the context of liquid and isomorphic software, the device management and orchestration are closely coupled. The common challenges include the following two key points:

- ensuring reliability and trustworthiness a distributed system where stateful components roam between locations
- minimizing disturbances to overall functionality while the software components behave in a liquid fashion.

Some of these issues have been addressed in our earlier work. Integration development and dynamic deployment to devices was address in [1] where IoT application development was addressed in DevOps spirit. The research challenges on managing Liquid AI applications have been discussed in [26]. In this paper we address installation of ML models similarly to traditional applications. In the future we plan to complement the system with full device management and orchestration functionalities.

3 Design and Implementation

3.1 Development Goals

The goal of our development approach was to investigate and hopefully demonstrate liquid deployment of AI/ML models on heterogeneous fleet of IoT devices. In a more technical sense, the target was two-fold:

G1: Demonstrate the feasibility of using WebAssembly to run AI/ML models on edge devices.
G2: Demonstrate the potential for liquid AI/ML software deployment on IoT devices.

The technical framing for the system was to use a microservice architecture consisting of various IoT devices building on our previous work [13]. Within this architecture, heterogeneous devices could easily be discovered and used in

accordance to their characteristics, such as varying processing power, available peripherals and ML/AI hardware functions, because application code could move inside the system. Computations would then be executed when and where best suited, taking into account the state of the system and the present capacity of the different nodes.

3.2 Implementation

As the baseline implementation technology we have used WebAssembly. This is the key enabling technology to achieve the free deployment of software modules across different hardware platforms, since as discussed, WebAssembly can be used as a runtime environment for applications developed using different languages, but compiled for the WebAssembly interpreter. It is also the technology we have used in our previous work, and the architecture to support liquid migration of functions followed the requirements identified in [13]. As a new challenge in the implementation, the applications that are deployed to devices include ML components.

To support freely moving code, a specific element in the architecture is a package-manager-housing orchestration server or *orchestrator*, whose functionality is depicted in Figure 3. While running, the orchestrator gathers 1) descriptions of devices available and 2) deployment *manifests* from users, that each hold the information needed for setting up an execution process/task as described in [13] where an application composed together from WebAssembly modules is run in the distributed system of heterogeneous devices.

The proposed system shown in Figure 3 consists of an orchestration server or *orchestrator* and a variable amount of heterogeneous node devices in the same local area network. An actor (user or another system that interacts with our system) can control the system through the orchestrator.

Aside from communication and application logic, the orchestration server consists of three components, presented below:

- *Device database* contains the hardware configurations of the various devices, and it is populated by listening to mDNS messages and requesting information from the associated devices.
- *Deployment registry* contains all executed deployments by the orchestrator, with each deployment listing the devices involved and the services they provide in it.
- *Package manager* maintains a database of all available WebAssembly software modules that can be sent to the devices. It is also capable of resolving dependencies to provide a complete list of required modules for a given module to run.

The system functionality can be split into three phases: device discovery, deployment and execution. Upon first discovery, the orchestrator requests configuration information from the device and adds it to the device database the server maintains. Upon a request for deployment, the orchestration server generates a setup that is a feasible deployment solution, and sends out the deployment

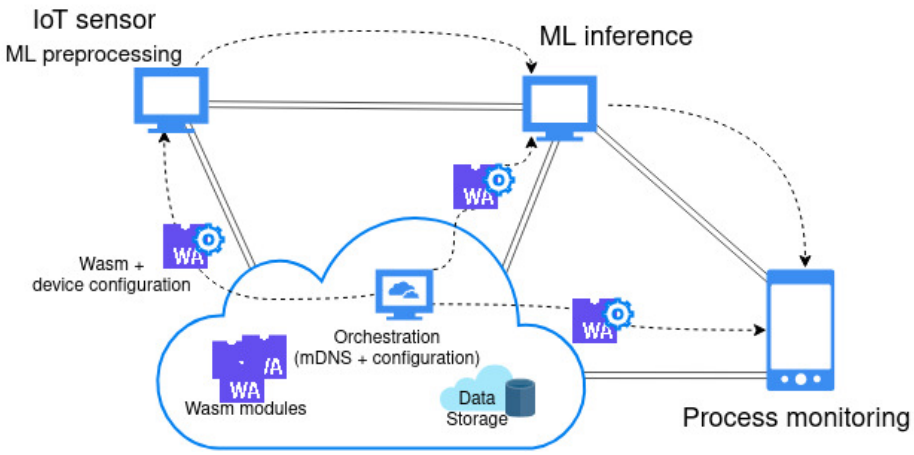

Fig. 3. Overview of the proposed system. In this example, an IoT sensor runs ML preprocessing on measurements, sending the results to an intermediary device with resources for ML inference based on the measurements. The results are then sent to a device with a screen for monitoring.

configuration to involved devices. The devices pull the required microservices and start serving them according to the deployment information.

Device discovery is performed with mDNS which each device uses to advertise their availability to the orchestrator. For querying the capabilities of discovered devices, a ReSTful endpoint providing the answers is available on each IoT device. ReSTful endpoints are currently also used for machine-to-machine (M2M) communication between the IoT-devices. A move to CoAP has been planned, to demonstrate that IoT specific protocols are feasible. Finally, all functionality running on the different IoT devices – in particular executing WebAssembly binaries – is controlled by the host process running on the device, which we call *supervisor*. The supervisor was implemented in Python, based on our earlier work, and the WebAssembly runtime for the moving code modules is Wasm3 [17].

The move to support AI/ML modules with our existing system necessitated changes to the ReSTful interface on the supervisor to facilitate upload of ML model files. Endpoints for uploading a pre-trained protocol buffer model file and running ML inference with supplied binary data were added. Additions were also required to the interface between the WebAssembly runtime and the host to enable loading of modules and input data to the linear memory of the runtime.

There are two ways to run ML models with WebAssembly. These are porting a ML framework to WebAssembly and running the model entirely in WebAssembly runtime, or outsourcing the ML model execution to a host-provided native ML runtime. As discussed earlier, latter approach has been formalized in a WASI proposal called wasi-nn, an extension for the WebAssembly System Interface to

provide an API to run ML models on a native ML runtime. The first experimental implementation of wasi-nn is for the Wasmtime WebAssembly runtime and uses the OpenVINO ML runtime. The experimental status of the implementation as well as lack of Python bindings for wasi-nn deterred us from starting with Wasmtime.

WasmEdge runtime implementation of wasi-nn has wider support for different ML runtimes including PyTorch, OpenVINO and Tensorflow-Lite, but Python bindings are a work in progress. Both Wasmtime and WasmEdge also have limited platform support compared to the Wasm3 runtime we have used in previous work.

Because of the current limitations of wasi-nn support addressed above, the supervisor prototype was done with ML framework compiled to WebAssembly, and run without a host ML runtime, using Wasm3 as the WebAssembly runtime.

The WebAssembly modules for AI/ML inference were written in Rust and the ML framework used was Tract[24], but these are of course interchangeable as long as the supplied model files are in the correct format for the used framework.

3.3 Results and Observations

We were able to realise a rudimentary system of orchestrating AI/ML applications across varied devices using WebAssembly.

We tested our system using pre-trained MobileNetV2 model for image classification. As expected using a framework compiled to WebAssembly results in poor performance. What was more surprising was the difference between different WebAssembly runtimes visible in table 1, as we also ran simple tests between Wasm3 and Wasmtime. Tests were run on a laptop with Intel Core i7-1165G7 and 16GB of RAM. The results highlight the disparate state of WebAssembly runtimes, but also the need for an extension like wasi-nn, as the pure WebAssembly approach will not be performant enough for all applications.

Table 1. MobileNetV2 execution times on tested runtimes, Wasmtime and Wasm3.

Runtime	Execution time (s)
Wasmtime	0.42
Wasm3	5.85

Building the system revealed a promising but still lacking framework for building liquid software systems with WebAssembly. The planned additions to WASI will likely alleviate the problems encountered in our implementation. Specifically standardized and extensive support for wasi-nn should enable an out-of-the-box solution for deploying AI/ML applications across different hardware platforms.

With the current state of affairs however, we encountered a myriad of issues in trying to implement our system. While some projects like WasmEdge were close to being useful, each had some drawbacks, such as narrow platform support or lack of bindings in our language of choice. While the lack of language bindings can in the WebAssembly ecosystem be mostly eliminated by using Rust, the lack of platform support or features like wasi-nn will likely continue to be a problem for some time.

4 Discussion

With the experience gained from our work so far it seems that the best supported language for working with WebAssembly is Rust, possibly owing to the fact that the reference runtime Wasmtime is implemented in Rust. The support for embedding WebAssembly in Rust as well as the tooling for compiling Rust to WebAssembly are more developed and better documented than Python, which we used for our supervisor implementation. Future development could benefit from moving to Rust as the development language. This could also ease turning developed functionality into contributions to existing WebAssembly ecosystem.

The lackluster performance of AI/ML inference on pure WebAssembly also motivates a move to wasi-nn-style paradigm of running the models, and will likely be necessary for wider adoption. This is however hindered by the lack of implementations in WebAssembly runtimes, but will hopefully improve as the wasi-nn proposal matures. As mentioned, currently the best support for running AI/ML models on host runtimes seems to be on WasmEdge runtime. Previously discussed move to rust would also allow us to change the WebAssembly runtime to WasmEdge, which would enable us to take advantage of WasmEdge's wasi-nn implementation.

In future we also hope to expand the functionality of the orchestrator. For example, the deployment requests need not be as specific as outlined above. The server could make decisions about device selection and deployment topology based on device availability and their dynamic state according to a deployment task describing desired deployment outcome without necessarily naming specific devices or their arrangement.

Including dynamic state for devices could also enable improved persistence and self-healing properties, as detection of failed devices could trigger a change in deployment topology and either a replacement device could be selected or the responsibilities of the failed device could be moved to another available device.

5 Conclusions

WebAssembly has been gaining attention outside the browser as a technique to speed up execution [4]. Its ability to support liquid deployment where applications can roam from one computer to another in an isomorphic fashion seems ideal for AI/ML applications that typically run in isolation, but may introduce strict requirements for performance.

In this paper, we have introduced a prototype system for isomorphic microservices based architecture for liquid deployment of AI/ML applications using WebAssembly, to test the limits of the above view. It was found out that executing AI/ML applications as WebAssembly modules in a WebAssembly runtime that there is a serious performance hit in comparison to running native code, and that running the applications using WebAssembly runtime requires both more working memory and persistent storage [9]. The runtime also complicates deploying applications that have real-time requirements, which are an integral part of many IoT use cases.

In the future, with increasing computational power and memory of IoT devices, the trade-off for ease of development and flexibility of deployment will likely become less and less of a problem for traditional applications. This in turn will liberate developers from considering some of the necessary practicalities during the development and deployment of services. Furthermore, in connection with AI/ML applications in particular, we expect that with emerging WASI extensions, such as wasi-nn, and host-bound specialised ML runtimes can be used to bring execution to near-native performance. However, this in turn can have impact on the isomorphic, liquid nature of the functions.

Acknowledgments.This work has been supported by Business Finland (project LiquidAI, 8542/31/2022).

References

1. Ahmadighohandizi, F., Systä, K.: Application development and deployment for iot devices. In: Lazovik, A., Schulte, S. (eds.) Advances in Service-Oriented and Cloud Computing. pp. 74–85. Springer International Publishing, Cham (2018)
2. Arnold, K., Gosling, J., Holmes, D.: The Java programming language. Addison Wesley Professional (2005)
3. Brown, Andrew and Mingqiu Sun: Neural Network proposal for WASI, `https://github.com/WebAssembly/wasi-nn`, retrieved 2023-05-09
4. Bryant, D.: WebAssembly outside the browser: A new foundation for pervasive computing. Keynote at ICWE'20, June 9-12, Helsinki, Finland (2020)
5. Bytecode Alliance: Welcome to WASI, `https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-intro.md`, retrieved 2022-12-05
6. Cloud Native Computing Foundation: WasmEdgeRuntime, `https://wasmedge.org/`, retrieved 2023-05-09
7. Gallidabino, A., Pautasso, C., Ilvonen, V., Mikkonen, T., Systä, K., Voutilainen, J.P., Taivalsaari, A.: On the architecture of liquid software: technology alternatives and design space. In: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA). pp. 122–127. IEEE (2016)
8. Gallidabino, A., Pautasso, C., Mikkonen, T., Systa, K., Voutilainen, J.P., Taivalsaari, A.: Architecting liquid software. Journal of Web Engineering pp. 433–470 (2017)
9. Hall, A., Ramachandran, U.: An execution model for serverless functions at the edge. In: Proceedings of the International Conference on Internet of Things Design and Implementation. pp. 225–236. IoTDI '19, Association for Computing Machinery, New York, NY, USA (Apr 2019). https://doi.org/10.1145/3302505.3310084

10. Hartman, J., Manber, U., Peterson, L., Proebsting, T.: Liquid software: A new paradigm for networked systems. Tech. rep., Technical Report 96 (1996)
11. Hartman, J.J., Bigot, P.A., Bridges, P., Montz, B., Piltz, R., Spatscheck, O., Proebsting, T.A., Peterson, L.L., Bavier, A.: Joust: A platform for liquid software. Computer **32**(4), 50–56 (1999)
12. Kotilainen, P., Järvinen, V., Tarkkanen, J., Autto, T., Das, T., Waseem, M., Mikkonen, T.: WebAssembly in IoT: Beyond toy examples. In: International Conference on Web Engineering. Springer (2023)
13. Kotilainen, P., Autto, T., Järvinen, V., Das, T., Tarkkanen, J.: Proposing isomorphic microservices based architecture for heterogeneous iot environments. In: International Conference on Product-Focused Software Process Improvement. pp. 621–627. Springer (2022)
14. Losant IoT, Inc: Embedded Edge Agent, `https://docs.losant.com/edge-compute/embedded-edge-agent/overview/`, retrieved 2022-11-09
15. Mäkitalo, N., Bankowski, V., Daubaris, P., Mikkola, R., Beletski, O., Mikkonen, T.: Bringing WebAssembly up to speed with dynamic linking. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. pp. 1727–1735 (2021)
16. Mäkitalo, N., Mikkonen, T., Pautasso, C., Bankowski, V., Daubaris, P., Mikkola, R., Beletski, O.: WebAssembly modules as lightweight containers for liquid IoT applications. In: International Conference on Web Engineering. pp. 328–336. Springer (2021)
17. Massey, S., Shymanskyy, V.: wasm3: The fastest WebAssembly interpreter, and the most universal runtime, `https://github.com/wasm3/wasm3`, retrieved 2022-12-09
18. Mikkonen, T., Pautasso, C., Taivalsaari, A.: Isomorphic Internet of Things architectures with web technologies. Computer **54**(7), 69–78 (2021)
19. Mikkonen, T., Systä, K., Pautasso, C.: Towards liquid web applications. In: International Conference on Web Engineering. pp. 134–143. Springer (2015)
20. Moreschini, S., Pecorelli, F., Li, X., Naz, S., Hästbacka, D., Taibi, D.: Cloud continuum: the definition. IEEE Access **10**, 131876–131886 (2022)
21. Peltonen, E., Ahmad, I., Aral, A., Capobianco, M., Ding, A.Y., Gil-Castineira, F., Gilman, E., Harjula, E., Jurmu, M., Karvonen, T., et al.: The many faces of edge intelligence. IEEE Access **10**, 104769–104782 (2022)
22. Rossberg, A.: Introduction — WebAssembly 1.1 (Draft 2022-04-05), `https://www.w3.org/TR/wasm-core-2/intro/introduction.html`, retrieved 2023-01-12
23. Rossberg, A.: WebAssembly Core Specification, `https://www.w3.org/TR/wasm-core-2/`, retrieved 2022-12-09
24. Sonos: Tract, `https://github.com/sonos/tract`, retrieved 2023-05-09
25. Strimpel, J., Najim, M.: Building Isomorphic JavaScript Apps: From Concept to Implementation to Real-World Solutions. O'Reilly Media (2016)
26. Systä, K., Pautasso, C., Taivalsaari, A., Mikkonen, T.: LiquidAI: Towards an isomorphic AI/ML system architecture for the cloud-edge continuum. In: International Conference on Web Engineering. Springer (2023)
27. Taivalsaari, A., Mikkonen, T.: A roadmap to the programmable world: software challenges in the iot era. IEEE software **34**(1), 72–80 (2017)
28. Taivalsaari, A., Mikkonen, T.: On the development of iot systems. In: 2018 Third International Conference on Fog and Mobile Edge Computing (FMEC). pp. 13–19. IEEE (2018)
29. Taivalsaari, A., Mikkonen, T., Systä, K.: Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. In: 2014 IEEE 38th Annual Computer Software and Applications Conference. pp. 338–343. IEEE (2014)

30. The OpenVino Project: OpenVino documentation, https://docs.openvino.ai/, retrieved 2023-6-13
31. The PyTorch Project: PyTorch web site, https://pytorch.org/, retrieved 2023-6-13
32. The TensorFlow Project: TensorFlow for Mobile & Edge, https://www.tensorflow.org/lite, retrieved 2023-6-13
33. The World Wide Web Consortium (W3C): Web Neural Network API, https://www.w3.org/TR/webnn/, retrieved 2023-6-13
34. Vetere, P.: Why wasm is the future of cloud computing, https://www.infoworld.com/article/3678208/why-wasm-is-the-future-of-cloud-computing.html, retrieved 2022-12-09
35. wasmCloud Project: wasmCloud home page, https://wasmcloud.com/, retrieved 2022-11-30