FH Salzburg
MultiMediaTechnology

Building and evaluating a WebAssembly based stream processing platform for a cooperative intelligent transport systems environment

Master thesis in partial fulfilment of the requirements for
the degree of

Master of Science

Author: Stefan Eisl

Submitted to the Master degree program MultiMediaTechnology, Salzburg University of Applied Sciences

Advisor: Andreas Bilke, M.Sc

Salzburg, Austria, 05.12.2022

Affidavit

I herewith declare on oath that I wrote the present thesis without the help of third persons and without using any other sources and means listed herein; I further declare that I observed the guidelines for scientific work in the quotation of all unprinted sources, printed literature and phrases and concepts taken either word for word or according to meaning from the Internet and that I referenced all sources accordingly.

This thesis has not been submitted as an exam paper of identical or similar form, either in Austria or abroad and corresponds to the paper graded by the assessors.

Date	*Signature*
	First Name *Last Name*

Kurzfassung

WebAssembly wurde ursprünglich entwickelt, um leistungsfähigere Anwendungen im Web zu ermöglichen. Seit 2017 wird es von den gängigsten Browsern unterstützt und es bringt die Eigenschaften mit sich, dass es schnell, sicher, hardwareunabhängig, plattformunabhängig, sprachunabhängig, effizient und kompakt ist.
Es hat sich gezeigt, dass genau diese Eigenschaften auch außerhalb des Browsers relevant sind. Daher wurde 2019 das WebAssembly system interface (WASI) vorgestellt, welches eine standardisierte Art und weiße bietet, wie WebAssembly sicher, schnell und auf nahezu jeder Plattform ausgeführt werden kann. Durch die Unabhängigkeit vom darunterliegenden System und der Eigenschaft, dass es sich immer mehr Programmiersprachen zu WebAssembly kompilieren lassen, könnte es für viele Szenarien eine Alternative zu traditionellen Containern darstellen.
Vorteile von WebAssembly im Vergleich zu einem Container sind unter anderem die Startzeit von wenigen Millisekunden, die Größe der Applikation, und eventuell auch die Geschwindigkeit. Es gibt bereits einige Benchmarks, welche die Geschwindigkeit von WebAssembly der von den in Container laufenden Applikationen vergleicht, allerdings handelt es sich hier nahezu ausschließlich um synthetische Benchmarks und keine realen Anwendungen.
In dieser Arbeit wird eine reale Plattform im C-ITS Bereich, zur eventbasierten Generierung von Warnungen für Verkehrsteilnehmer, mittels WebAssembly Modulen implementiert. Die Schwierigkeiten sind hierbei, dass die Technologie noch sehr jung ist, es eine nur sehr begrenzte Auswahl an Frameworks gibt, welche für den Einsatzzweck verwendet werden können. Bei der Plattform handelt es sich um eine eventbasierten Stream Processing Pipeline, welche mittels Spring Cloud Streams und Kafka Streams Microservices aufgebaut ist.
Teile der Plattform werden in den drei WebAssembly Frameworks, Fermyon Spin, WasmCloud und Spiderlightning implementiert und mit der originalen Plattform verglichen. Wobei die relevante Metrik die Laufzeit der gesamten Plattform ist. Es zeigt sich, dass sich die Laufzeit für die gesamten Verarbeitungsschritte je nach Framework stark unterscheiden. So ist die schnellste WebAssembly basierte Implementation je nach Szenario zwischen drei und 67-mal schneller als die Referenz und dabei weniger Ressourcen benötigt. Andere hingegen zeigen vergleichbare Ergebnisse, oder sind deutlich langsamer. Insgesamt kann gezeigt werden, dass sich mit den aktuell verfügbaren Werkzeugen, eventbasierten Stream Datenverarbeitung realisieren lässt. Wobei es je nach gewähltem Werkzeug zu unterschiedliche Einschränkungen kommt. Es kann gesagt werden, dass WebAssembly eine gute Basis für ein Streamdatenverarbeitungs-Framework bilden kann.

Schlüsselwörter: webassembly, wasi, wasm, streamprocessing, c-its, Spin, Slight, Spiderlightning, WasmCloud

Abstract

WebAssembly was originally developed to enable high-performance applications on the web. Since 2017, it is supported by the most popular browsers, and it brings with it the characteristics of being fast, secure, hardware-independent, platform-independent, language-independent, efficient and compact.
It has been shown that exactly these properties are also relevant outside the browser. Therefore, in 2019, the WebAssembly system interface (WASI) was introduced. It provides a standardized way for WebAssembly bytecode to run securely, quickly, and on almost any platform. Due to its independence from the underlying system and the fact that many programming languages can be compiled to WebAssembly, it could be an alternative to traditional containers for many scenarios.
Advantages of WebAssembly compared to a container are the startup time of only a few milliseconds, the application size, and possibly the execution speed. There are already some benchmarks comparing the performance of WebAssembly to applications running in containers, but these are almost exclusively synthetic benchmarks and not real applications.
In this work, a real-world platform in the Cooperative Intelligent Transport Systems (C-ITS) domain, for event-based generation of alerts for road users, is implemented using WebAssembly modules. The difficulties here are that the technology is still very young and there is a very limited choice of frameworks that can be used for this purpose. The platform is an event-based stream processing pipeline built from microservices using Spring Cloud Streams and Kafka Streams.
Parts of the platform are implemented within the three WebAssembly frameworks, Fermyon Spin, WasmCloud and Spiderlightning and are compared to the original platform. The relevant metric for the C-ITS use case is the latency of the entire platform. Within this work, it can be seen that the latency for the entire processing steps differ greatly depending on the framework. For example, the fastest WebAssembly-based implementation is between three and 67 times faster than the reference, depending on the scenario, while also using fewer resources. Others, however, show comparable results, or are significantly slower. Overall, it can be shown that event-based stream data processing can be realized with the currently available tools. Whereby it comes depending on the selected tool to different restrictions. It can be said that WebAssembly can form a good basis for a stream data processing framework.

Keywords: webassembly, wasi, wasm, streamprocessing, low-latency, c-its, Spin, Slight, Spiderlightning, WasmCloud

Contents

List of Figures

Listings

List of Tables

1 Introduction

Randall (2021) from the Cloud Native Computing Foundation (CNCF (2022)) writes, that within the last 15 years, there were several evolutions when it comes to cloud computing. One of them was the introduction of public clouds, like Amazon AWS or Microsoft Azure. This brought the ability to run virtual machines in the cloud and with this a decoupling from the maintenance and managing of server hardware and the software took place. When Docker came out in 2013, and Kubernetes was released in 2014, the usage of container-based applications, instead of using entire virtual machines, has been another evolution. This brought another layer of abstraction, where developers only have to package applications and the needed dependencies in containers and deploy them.
Now, according to a survey by the Cloud Native Computing Foundation, WebAssembly might be another evolution for cloud computing. Especially in the fields of serverless and edge computing.
WebAssembly (Wasm) was initially designed to run within a browser, it was intended to be a small binary, secure and fast starting and language neutral compilation target for different programming languages(*WebAssembly specification* 2022). It can be used inside the browser alongside JavaScript for handling computational heavy workloads, or to enable the usage of various programming languages including C#, C++, Rust[1], Go[2], Python, AssemblyScript[3] and others[4]. Since the defined goals for Wasm are also desirable for applications outside the browser, in 2019, a standardized interface, the WebAssembly System Interface (WASI), has been introduced, which enables, to run WebAssembly modules outside the browser, on almost any platform, in a save, sandboxed environment(Clark 2019b). This makes it an alternative to container runtimes like Docker or CRI-O. Furthermore, since cloud instances, which are running on CPUs based on the ARM architecture, instead of x86, Wasm has shown to be even more platform-agnostic than traditional containers. A container needs to be built for a specific CPU architecture while Wasm bytecode, can be executed on x86 or ARM as well as others(*WebAssembly specification* 2022). According to Gackstatter (2021) it also promises to provide a higher throughput and a minimal startup time and according to Yuan (2021), the performance of an application compiled to Wasm is better than the native application within a docker container. Since WebAssembly is gaining traction, and is getting more mainstream, tools, platforms, and frameworks emerged, which are trying to make the use of Wasm for server-side applications practicable. Although, according to the survey from the CNCF (2022), the missing tooling is still one of the main barrier for using WebAssembly.

Within this thesis, the capability of Wasm utilizing the WebAssembly system interface will be evaluated on an actual streaming data processing use case, within a research project in the field of Cooperative Intelligent Transport Systems.

The goal of the research project is a system that should reduce traffic accidents of vulnerable

1. https://www.rust-lang.org/ visited 11.11.22
2. https://go.dev/ visited 11.11.22
3. https://www.assemblyscript.org/ visited 11.11.22
4. https://webassembly.org/getting-started/developers-guide/

road users using vehicle-to-everything (V2X) communication and is called Bike2CAV[5]https://www.bike2cav.at/en/home-2/. Within this research project, roadside units (RSUs) which can receive messages from the vehicles, have been installed at crossroads. Those RSUs are also recording the positions of non-connected road users. The recorded messages are then sent to a cloud service, which is used for the processing of the recorded information. The data sent to the cloud services are high frequent Collective Perception Messages (CPMs)(ETSI 2021) which are a part of the Cooperative Intelligent Transport Systems (C-ITS) standard from the European Telecommunications Standards Institute (ETSI). Those messages then are getting analyzed and warnings in the form of Decentralized Environmental Notification Messages (DENMs)(ETSI 2019) are returned to possible affected vehicles, either directly over the internet or to the RSU which then forwards it to the vehicle.
The objective of this thesis is to evaluate the capability of server-side WebAssembly in building a stream data processing platform, which creates warnings from the data gathered by the RSUs. This task consists of the following processing steps.

- Extracting the geospatial information from the CPMs and store them in a unified messaging format for location-based information
- merging the location-based messages to trajectories
- Mapping of the trajectories to a street graph which is aware of the semantics of the road
- Predicting possible paths according to the semantics of the street graph
- Generate warnings for possible collisions from the predicted paths from the current road users
- Wrap the warnings into the DENM format and send them to a message broker.

As reference implementation for the WebAssembly-based approaches, a stream processing platform which consists out of a message broker, which receives messages from the connected vehicles or RSUs, an Apache Kafka as streaming platform, where the messages are appended to a persisted message queue, called log is used. Attached to those event streaming platform, there are general purpose microservices, which are consuming messages from the log, applying transformations or aggregations, and then passing the messages to another topic of the log or a sink. In the reference approach, which is using microservices, every service only does a single transformation or aggregation. This approach has been chosen because the results of the single processing steps are also used within other projects (1). The usage of WebAssembly within a stream processing pipeline would lead to the following benefits to the selected use case.

Possible serverless The platform could be serverless, where services are only running when they are used. This could result in a reduction in power consumption, as during less busy hours at the intersection, there are only a few messages sent to the system.

5.

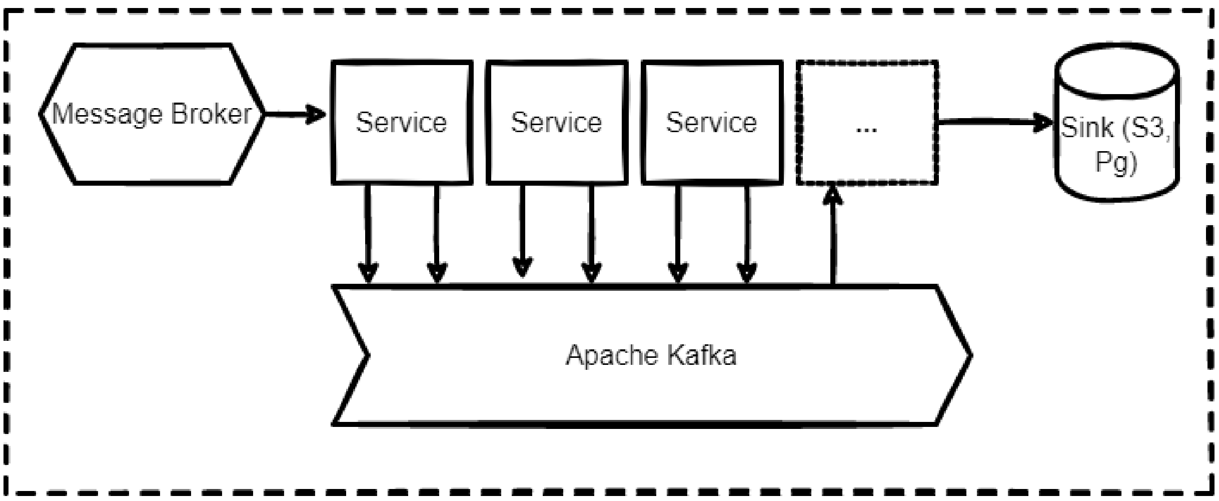


Figure 1: Simplified sketch of a microservice approach

Fast Since Wasm promises near native performance, it should be faster than containerized applications for this use case.

Portable Individual services can be deployed on edge devices as well as cloud servers.

Language agnostic Developers can choose the programming language they prefer when building services for the platform. This benefit is especially interesting, since Python can be compiled to Wasm (Reese and Butcher 2022), and it's a popular language for data science (M. Tim 2018). For the actual research project, data scientists are also utilizing the streaming platform for their research.

The question is, if it is possible to implement an event-driven stream data processing pipeline with the use of WebAssembly modules with the available tools and frameworks, and if possible performance benefits of WebAssembly over containerized applications can be translated to the task of processing high frequency real-time data.

To answer this question, the microservice-based processing pipeline for a C-ITS use case is compared with multiple Wasm-based approaches. One WebAssembly-based approach is serverless, and the others are using long-running services. The metrics which will be considered are the throughput, the service-to-service latency, the end-to-end latency and the processing time of the individual services.

2 Methods

Within the methods, WebAssembly will be described, especially from the cloud computing perspective. Then stream processing in general will be discussed, followed by related research regarding those topics. Then an overview of available WebAssembly frameworks and platforms is presented. The last part of this section is a detail view on the actual data streaming platform is presented with a detailed view on three particular services.

2.1 WebAssembly

> WebAssembly… is a safe, portable, low-level code format designed for efficient execution and compact representation. Its main goal is to enable high performance applications on the Web, but it does not make any Web-specific assumptions or provide Web-specific features, so it can be employed in other environments as well. (*WebAssembly specification* 2022, 1)

The development of WebAssembly has been announced in 2015, and a first version was released in 2017. Although, this first released version was only a minimum viable product (MVP). The WebAssembly standard is open and developed by WebAssembly community group of the W3C[6]. It came with the goals of being fast, with a near native performance, safe, where the code is executed within a sandboxed environment and portable, where the code can be executed on any modern hardware architectures and environment. Other goals were efficiency-driven, where the binary should be compact, modular, efficient, and streamable.(*WebAssembly specification* 2022, 1) The list of goals is not exhaustive, and the three for this thesis most relevant aspects are further described. Those are the performance, the hardware-, and platform-independence and the security.

performance As the results from Bhonsle et al. (2022, 5) are showing, the computation performance of Wasm is between 10 times and 50 times faster than JavaScript for linear algebra workloads like matrix multiplications. This example demonstrates, that the goal for faster code execution can be met, at least when compared to JavaScript inside a browser environment. As for this thesis, not the performance difference between Wasm and JavaScript is relevant, but the performance in comparison to containerized or native running applications is is. According to Long et al. (2021, 4), the performance of Wasm is between 10 and 50% better than the same code executed within a Docker container. In that comparison, C code that has been compiled to WebAssembly was compared to the same piece of code executed within a container. Yuan (2021) did a performance analysis on Arm and x86 CPUs, where he also included a JavaScript implementation of an application to his comparison. For the benchmark the nbody, the fannkuch-redux, the mandelbrot, and the binary-trees test from the Computer Languages Benchmarks Game[7] have been evaluated. The result was that Wasm within the SSVM runtime, this runtime is now called WasmEdge, is up to 5 times faster tan the JavaScript implementation and significant faster than the containerized native applications for smaller workloads, while being slightly faster for computational heavy ones.

6. https://www.w3.org/community/webassembly/
7. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

portability Accoarding to the *WebAssembly specification* (2022, 1–2), Wasm is designed to be hardware-independent, language-independent as well as platform-independent. Hardware-independent means, that the same Wasm binary can be executed on systems which are using an x86 as well as an ARM, or RISC-V CPU architectures, or even embedded systems like the ESP-IDF platform.
The term language-independent means, in this context, that many programming languages can be compiled to WebAssembly. Some of them are C#, C, C++, Ruby, Go, Python, Rust, Haskell, AssemblyScript and others. However, the maturity of the ecosystem varies greatly. For server-side applications, C/C++, Rust and Go seems to be the languages that are mainly used.
With platform independent is meant that it can run within a browser, in an own runtime, or it can be embedded within another application.

security Clark (2019a) explains that the implemented security for Wasm is, that it provides no access to the underlying computing environment. This means that the WebAssembly module cannot use the file system, system calls or I/O by its own. To be able to access those capabilities, the system which embeds the Wasm module, have to provide those as functions to the module, and the module needs to import those. Passing functions is currently done, by passing a file descriptor. This way, the platform can define the security policies, determining which permissions embedded applications have or not. Wasm itself uses a linear memory model, which is an array of raw bytes, that is mutable and can grow dynamically. Operations are only possible within this array of bytes. When reading or writing data, there is also a check if the access is within the given bounds of the linear memory, if it is not, the execution of the application is instantly aborted, and can be handled by the platform which embeds the Wasm module. Lehmann, Kinder, and Pradel (2020) found several security vulnerabilities within their research. Most of the found vulnerabilities origins are within the handling of the memory. Although when using a memory safe language, when writing the applications, as well as only importing the capabilities, that are actually needed, most of the vulnerabilities are resolved (Lehmann, Kinder, and Pradel 2020, 231). Gackstatter (2021, 16) noted that those vulnerabilities can only affect the Wasm module itself, but not the host system, in which it is executed. Since the system which is embedding the Wasm modules is responsible for the capabilities of the executed modules, an as permissive strategy for granting capabilities will also reduce the vulnerabilities. For example, when access is only needed for a single file, only an opened file descriptor is passed to the module for that specific file(Clark 2019b).

2.1.1 Wasm in the browser

When WebAssembly is used within the browser, as originally intended, according to, Clark (2019b) the browser acts as the runtime for WebAssembly. Since 2017, it has been supported by the four main web browsers, which are, in alphabetical order, Apple Safari, Google Chrome, Microsoft Edge and Mozilla Firefox. To make use of the web APIs, provided by the browser, JavaScript functions can be exported and imported into the Wasm module. Since those web APIs are standardized, this works for any of the main browsers.

2.1.2 Wasm outside the browser

There are several ways to run WebAssembly modules outside the browser. One of the first approaches was, to utilize the V8 JavaScript and WebAssembly engine, which is also used in Node.js(OpenJS Foundation 2022). To achieve this, the WebAssembly module can be read and executed from the node.js server. Within this thesis, the WebAssembly modules should be used without any dependencies for Node.js and others. Therefore, using Node.js is not viable. Besides Node.js there are other ways to execute Wasm modules outside the browser. There are already several WebAssembly runtimes available, which differ in their implementation and their functionality. The runtimes which are used within the frameworks, that are discussed within this thesis are WasmEdge by Second State[8], Wasmer[9] and Wasmtime by the Bytecode Alliance[10]. Those three are general purpose runtimes, but there are also ones for specific environments like microcontrollers, or real time operating systems (Second State 2022b). They differ in various ways, but they have the usage of WASI in common. This interface, as well as an excerpt of the runtimes, with their abilities, are briefly explained.

2.1.3 WASI

According to Hickey et al. (2020a), the WebAssembly system interface is a standardized set of APIs that is implemented within the Wasm runtimes. This interface enables that applications can be run on any runtime which implements those. It should be noted that a runtime does not need to implement the entire interface, but can also only use a subset of the APIs. This results from the diversity of environments, a WebAssembly runtime might be used. As an example, the system interface of neural networks might not be needed within runtimes, that are targeting microcontrollers, so it does not need to be implemented. The main goals of WASI are security, portability, interposition, and compatibility. It is also intended to be usable within the browser, where there is an existing polyfill available, that maps the calls from the Wasm module to the web API from the browser. The implementation method for the goals is explained further.

Security Security is implemented by a so-called capability-based security. Like already in 2.1 mentioned, external resources, such as files, can only be accessed, if the capability is passed to the application from the runtime. Those external resources are passed in the form of handles, which are unforgeable values that represent the resource. Those handles also cannot be requested by, they always have to be provided. With this kind of security implemented, a malicious application, can be executed safely, since there is no way, it can access resources like system calls or files, that are not provided by the runtime(Clark 2019b).

Portability Although portability is one of the main goals, depending on the system the implementation is used for, it is sometimes not possible to achieve a 100% portable API. Therefore, the committee decides for each API if it fits the goal.

8. https://www.secondstate.io/ visited at 25.11.22
9. https://wasmer.io/
10. https://bytecodealliance.org/

> We prefer APIs which can run across a wide variety of engines when feasible, but we'll ultimately decide whether something is "portable enough" on an API-by-API basis. (Hickey et al. 2020a)

Interposition Interposition means that a WebAssembly module can import an interface which might be exported by another module. The actual resolution of the implementation happens at the instantiation of the module. This leads to the behavior, that the actual implementation of an interface can be changed, while the module which uses it would not notice this.

Compatibility WASI tries to keep its APIs free of capability concerns. This is done via, moving those things to the libraries. This enables, that only applications, which are actually using such libraries are affected. Those compatibility relevant parts are for example in the `WASI libc`, which is a set of libraries, that is used instead of the actual `libc`, when the compilation target of a code that is compiled with the `wasm32_unknown_wasi` target triple.

2.1.4 Proposals

Adding new features, or changing existing ones, is only possible via a proposal(Hickey et al. 2020b). Such a proposal can be initialized, with the creation of a GitHub issue in the design repository of Webassembly[11]. This issue is then discussed, and eventually added to the proposal list. From there, there are five more phases to go through. In the first phase, the design of the feature needs to complete. In the second phase, one or more implementations for a prototype, as well as a test suite needs to be finished. In the third phase, the feature is implemented by at least two runtimes. In the fourth phase, the feature is passed to the official WebAssembly Working Group. They discuss edge cases, and can confirm if a feature is complete. Only minor cosmetic changes are done in this phase. If rework is needed, it goes back to phase three. In the final phase, the feature is seen as complete, and the repositories are merged.

2.1.5 Runtimes

As mentioned at 2.1.2, there are several runtimes for WebAssembly modules. They might differ in things like the operating system they can run on, the way they are executed, the hardware platform, the language in which they are written and if they have implemented future proposals of the Wasm standards. Since there are too many runtimes to cover them all, only the ones from the evaluated frameworks will be investigated with the addition of the WebAssembly micro runtime (WAMR) and wasm3.

Wasmtime Wasmtime is from the Bytecode Alliance and written in Rust. The Bytecode Alliance was founded by Mozilla, Fastly, Intel, and Red Hat(Clark 2019a). The focus of Wasmtime is security and corretness(Bytecodealliance). It does come with a command

11. https://github.com/WebAssembly/design/issues

line interface (CLI) from which Wasm modules can be executed, but can also be embedded into Rust, C, Python, .NET and Go. The runtime can be used on the arm64, as well as x86 and RISC-V CPU architectures on Windows as well as Linux and macOS. It supports the entire WASI API including most of the proposals, if it is likely that they got excepted(Bytecodealliance).

Wasmer[12] Wasmer(Wasmer, Inc 2022) comes, like Wasmtime with a CLI and can be embedded within multiple languages including Rust, C++, Go, Python or Ruby. It runs on the same operating systems as Wasmtime and supports the same CPU architectures. Also like Wasmtime it is written in Rust, and many of the Wasm proposals are implemented. A difference to Wasmtime is, that there are multiple compiler options available, for faster compilation speed or runtime speed.

WasmEdge WasmEdge, former SSVM, by Second State differs from the mentioned runtimes in several ways. One difference is, that it can run modules in interpreter mode instead of compiling them. Although, in this mode the runtime speed is much slower as if the module is compiled ahead of time (Second State). can run applications in interpreter mode and in AoT mode, in which case the interpreter mode is much slower(Second State). Another difference is, that it is written in C++ instead of Rust. It can also be used on Android devices (Second State 2022b). Another difference is the implementation of additional features. While the other runtimes stick to the WASI standard, WasmEdge has proprietary extensions to it. This leads to capabilities, which are not available for other runtimes. Such features are network sockets, image processing libraries and more (Second State 2022a). The official WASI API is also implemented. The additional features can be utilized, through an SDK provided by WasmEdge for C/C++, Rust and Go.

Wasm3 Wasm3(Shymanskyy) does only support an interpreter mode. The reason for this is, that this runtime puts more weight in compatibility than in runtime speed. The interpreter approach leads to smaller executable sizes, less memory usage and a smaller startup latency. The minimal system requirements are about 64 Kb for the application itself, and about 10 Kb of ram. Wasm3 can be used on several architectures, and additional to the three big platforms, Linux, MacOS and Windows, also on microcontrollers like Adrduino, ESP8266, ESP32 and more. It can even be used within WebAssembly itself. Wasm3 is written in C, and can be embedded into languages as Python, Rust, Perl, Swift, .Net, C/C++ and others. Like WasmEdge it has some additional capabilities, which are not part of WASI. An example for this is gas metering, which limits resources for an executed module.

WAMR The WebAssembly Micro Runtime is another runtime from the Bytecode Alliance and was initially created by intel[13]. It can be used with an interpreter mode, compiled ahead of time and just in time. When using in interpreter mode, it is possible to use it on several platforms, including android and microcontrollers like the ESP32. Like with its peers, the usage of the interpreter mode is limiting the execution speed. It is written in

13. www.intel.com

C and can be embedded into Pyton, C/C++ and Go. Features from the Wasm proposals are also built in this runtime. When using WAMR, it can be decided if the standardized WebAssembly system interface, or a proprietary interface is used for executing the Wasm modules(Bytecodealliance 2022).
When using the proprietary libraries, additional features like multithreading are available, although the security model which is implemented with WASI is not utilized, which means, that capabilities do not need to be passed to the WebAssembly module for it to utilize it. For example, the WebAssembly module will be able to open files on its own. WAMR uses, similar to WASI, a modified version of the standard libraries for the implementation of is proprietary features. Therefore, if the WASI standard is going to implement a feature, which is now only available through the WAMR libraries, the modules should be compatible.

From this excerpt of runtimes, it can be observed, that every runtime utilized the WASI standard. Some of them are extending it, as it might be too restrictive for various applications. When using the extended features, the compatibility with other runtimes, is not given anymore, although for the WAMR implementation, which uses the same approach as WASI, with replacing `libc` with a different one, the compatibility should be available, as soon as WASI implements this feature. For example, there is a proposal for multithreading in WebAssembly(Rossberg 2022), which might be implemented by the runtimes. As soon as this happened, a Wasm module which uses the `pthread` library, which is currently only implemented in the WAMR runtime, can also be used by the runtimes, that have implemented this new proposal.

2.1.6 The Component Model

Currently, there is a proposal which is, according to Jiaxiao (2022) form Deis Labs research [14], as well as Taylor (2022), Dice (2022) and others, whom are part of different frameworks that are utilizing WebAssembly, the way WebAssembly should be used if possible. This proposal is the so-called component model. It introduces a way multiple Wasm modules can interact with each other, or with outside interfaces which are not implemented in WebAssembly. It is briefly explained because, two of the evaluated frameworks are using already parts of it, and another one has mentioned in a blog post, that they are going to make a major refactoring of their framework, so they can utilize the benefits of this proposal.
The component model is a way to add different WebAssembly modules together, which can import and export functionality. In addition to the exported functionality, there are interfaces defined. The exporting module has to implement the functionality of the interface, and the importing module can utilize those functionalities. The interface is defined in the form of a `wit` file, which describes the functions or members of an interface. A minimal example of such an interface is shown in 1.

```
use { error, payload } from types
resource kv {
    // get a payload for key.
```

14. https://deislabs.io/ visited 11.11.22

```
    get: func(key: string) -> expected<payload, error>
    // set the payload for a key.
    set: func(key: string, value: payload) -> expected<unit, error>
}
```

Listing 1: minimal key value store example interface in wit

Currently, this proposal is not implemented entirely. Although it is already possible that interfaces are imported into Wasm modules and exported from native binaries compiled by Rust or C/C++ using the wit-bindgen(Alliance 2022) project. The frameworks, which are evaluated in this thesis, which are utilizing parts of the component model are doing exactly this. As a simplified example, the interface from 1 is defined. an application, which wants to use those functions, imports only that interface. This module, called guest module, can now be executed by any Wasm runtime, that exports this interface. The host runtime, on the other side, implements and exports the interface. When a guest module is initiated by the runtime, the module gets linked to the interface the host is exporting. The guest module can now invoke the function, which is actually executed within the host runtime.
Hoffman (2022) states, that the component model will lead to a better decoupling from the non-functional requirements and the actual business logic. Such a decoupled approach is already implemented in the Spiderlightning 3.3 project, as it offers multiple implementations for a couple of commonly used interfaces, such as pub/sub brokers, key value stores or blob stores. Since the component model is not yet final, and is therefore not implemented entirely, the provider of the exported components is a native executable instead of a Wasm component(Jiaxiao 2022).
The component model is similar to common dependency injection, but offers the unique feature, that the different components can be built in different languages. Therefore, a component could be written in Rust, Go, .Net or any other language that compiles to Wasm, and they would still be compatible. In addition, the security mechanism from WASI is still utilized, which reduces the vulnerability for supply chain attacks, when components are used from a package manager like NPM(Reisz and Clark 2021). Reisz and Clark (2021) mentioned, that the pattern is similar to the ones from Jakarta Enterprise Beans, which also offers standardized components for Jakarta EE, former Java-EE[15].

2.2 Stream data processing

Stream data processing, is the processing of data within an ongoing sequence of messages, as they are received. The ongoing sequence of messages is called a stream, and it has no bounds. Such a stream can consist of messages, which are sent at various frequencies, and could also consist of only a single message. For stream processing, tasks are applied to every incoming message from the stream. This is different from batch processing, where data is collected for and processed afterwards. While the batch processing might be more efficient, stream processing is needed if an instant feedback is required.

15. https://jakarta.ee/

2.2.1 Common usecases

There are several fields, for which stream data processing is used. A small excerpt is briefly describing use cases by an example.

Fraud detection Moffatt (2018) shows, in his article, how fraud detection can be done with the usage of the Apache Kafka Stream processing engine. In his example, the transactions from ATMs are taken as the messages of the stream. For every message of the stream things like the georeferenced coordinates of the transaction, and the timestamp are compared with previous ones from the same account ID. If the geographic distance is large between transactions of the same account, and the time between them is too short, a possible fraud is detected, before the transaction is finalized.

e-commerce Within the e-commerce branch, stream processing is used, to track the behavior of the users, while those are using the online shop. In this case, the interaction of the user within the shop are the messages of the stream. An example could be, that a user adds something to the basked, but ends the session, and therefore there have been no checkout. A simple approach for the stream processing engine could be, to open a time-based session window for the user, which is closed after a defined time after the last message for this session. If the window gets closed, and there was a message for adding something to the cart but none for removing or purchasing this product, an event could be triggered.(Dean and Crettaz 2019)

Transportation In the field of intelligent transportation, there are several use cases for stream processing. Some of these use cases are the turn-by-turn navigation, counting of vehicles, the calculation of traffic conditions. An example for a transportation use case can be the generation of traffic warnings. In such a use case, smartphones of the street users could send at a specific interval messages, containing its location. Those messages are representing the stream. The messages of the stream can be aggregated by the individual devices, and the movement speed can be calculated by using the distances between the contained location information with the timestamps. If there are many slow-moving devices on the same street segment, a warning for slow traffic is generated.

2.2.2 Common functions

For stream processing, there are several functions which can be applied to the messages. An excerpt of such functions is how they differ to each other.

Transformations Harsh (2022) states, that data transformation is used, when the format or the schema of a message needs to be changed. Such a function could parse a date which is represented as string and transforms it into an actual data format. Another typical use case is, if the structure of the message needs to be adjusted, or if it needs to be encoded, or decoded. Transformations are usually stateless, as they do not require information from previous messages.

Filters Filters are functions which discard messages, which does not meet some defined behavior. An example could be, a filter that discards messages, where a value is not given or out of a specified range. Like the transformation, filters are also usually stateless.

Aggregations According toHarsh (2022), aggregation of data, is when messages for a specific identifier are combined into an aggregate, and algorithms are applied to this set of messages. Aggregating can be used to find outliers, merging data, counting and more. This function is stateful. Aggregation of data is usually done within some kind of time-based windowing, as otherwise the aggregates would get too big. Some possible windowing methods are.

Tumbling Tumbling windows are a fixed time range for which data is aggregated. This means, that if this window is defined with one hour, all messages are aggregated for an hour, after this hour messages, which would be added to an existing aggregate, will be aggregated as new set of messages.(Microsoft 2022)

Hopping Hopping windows do have a fixed time range like the tumbling window, but they hop forward, in a separate defined amount of time. This way, messages can be aggregated in more than just one window.(Microsoft 2022)

Session For the session window, messages are aggregated, as long, there are messages for its identifier. There is also an inactivity gap, which defines the period, for how long the window will be open after the last received message. As an example, within an online shop, every interaction of a user lead to a message. If the user stops shopping, the session will be closed after a defined period. When the same user goes to the shop again, a new aggregate will be generated, as there is no open session window for this user.(Microsoft 2022)

Sliding For the sliding window, messages are aggregated within a defined timestamp before the message arrived. This means, when a message arrives, every message with the same identifier, which has been received within a defined period before the message, is aggregated. A possible use case would be the calculation of an average value for the last period of time.(Microsoft 2022)

Enrichment The enriching of messages, is when additional resources or given data is combined with the message to add value to the messages(Harsh 2022). An example for such an enriching would be the adding of user-related data to a message, by getting those from a database with the identifier from the received message.

Analytics Analytics means the analysis of the messages from the stream. This information can be used for visualization, alerts, and other.

2.2.3 Tools and frameworks

There are several tools available for stream data processing. According to Chintapalli et al. (2016, 1) the most popular ones are Apache Storm[16], Apache Flink[17] and Apache Spark[18]. Therefore, these are briefly described. The streaming engine Apache Kafka as well as Kafka Streams, and Spring Cloud Streams are also described, since it is utilized within the reference implementation of the stream processing pipeline.

Kafka Apache Kafka is a distributed streaming platform which persists every transaction to the file system. It can also be used as publish/subscribe message broker that provides a guaranteed exactly once delivery for the messages. Data is stored in form of messages in order and can be replayed. Apache Kafka also promises high throughput combined with a low latency. The Apache Kafka streaming engine consists out producers, consumers, nodes, and topics. The producer is a client which sends messages to a topic. A consumer reads from a topic. A node is an instance of Apache Kafka, the topics are contained within the nodes, and multiple nodes can contain the same topic. A topic is a message log which contains the messages. The topic itself is split into partitions, where every partition is again a unique log and can only be consumed by a single consumer. The broker keeps track of messages that have been read by storing the offset of the latest consumed message. This offset is stored for every consumer group, which means if a new consumer with an unknown consumer group wants to read messages, it will start at the beginning of the log. This concept allows having multiple consumers reading from the same topic without reading messages multiple times. It is also possible to consume from the broker without increasing the offset at the broker, but this is not relevant for this evaluation.
Unlike publish and subscribe message brokers, Apache Kafka uses a pull-based messaging strategy. This means that the consumer has to actively pull the messages from the topic. Benefits of such an approach are that multiple consumers can consumer with different pace or even allow that one consumer can consume messages in batches, while others receives them one by one. Kafka also offers the Kafka Streams2.2.3 data streaming framework. In current versions of Apache Kafka, there is an Apache Zookeeper[19] instance necessary per Apache Kafka node. Apache Zookeeper is used for the leader election of the Kafka nodes and for the configuration of the topics access control lists. Future versions of Apache Kafka will also be able to run without Apache Zookeeper.

Spring Cloud Streams Spring Cloud streams is a Framework by Spring[20]. Sabby et al. (2022) writes within the official documentation. Spring Cloud Stream serves as an abstraction to Spring Kafka and Kafka Streams2.2.3 applications. It can be used for building microservices which are connected with messaging brokers or streaming engines like Apache Kafka. It decouples the configuration to the messaging system and the actual business

16. https://storm.apache.org/
17. https://flink.apache.org/
18. https://spark.apache.org/
19. https://zookeeper.apache.org/ visited 11.11.22
20. https://spring.io/

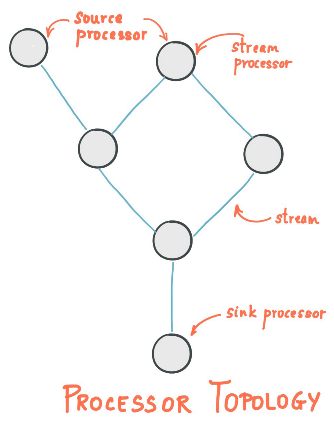


Figure 2: The topology from Kafka Streams (Apache Foundation 2022c)

logic from the application. The setup for the connection to the broker is completely done within the configuration file and the without writing any code. Spring Cloud Streams enables to compose multiple processes, with each other, within one application. This function composition allows applying multiple stream processing tasks on the stream, without producing and consuming the messages from the broker between the tasks. The functions which are applied can also be defined within the configuration and can be changed at runtime. Once the functions are configured, they are applied on every consumed message from the Apache Kafka broker[21]. Spring Cloud Streams can also be used with the Kafka Streams library. In this case, Spring Cloud Streams takes care of the connection to the Kafka Streams capabilities can be utilized by the native Kafka Streams implementations.

Kafka Streams As stated in the documentation by Apache Foundation (2022c), Kafka Streams is specifically designed for stream data processing, where the input, as well as the output data, is stored in Apache Kafka topics. It utilized its topics on the Apache Kafka broker to keep its state persistent and distributed, as well as an in RocksDB for its local state. The RocksDB allows storing large amounts of data as state because it writes the data to the disk instead of the memory. In contrast to the normal Kafka Consumer, Kafka Streams offers a high-level domain-specific language (DSL) that provides APIs for common processing tasks like windowing2.2.2, filtering2.2.2, aggragating2.2.2 and transformations2.2.2. It also offers a low-level API, that enables more customizable functionality. The topologies of Kafka Streams looks like figure 2. It consists out of one or more source processors, one or more additional stream processors, the stream, and a sink processor. The source processor produces the stream from a Kafka topic, the stream processors are applying the processing on the messages of the stream, where there is always one stream processor per processing task, finally the sink processor writes the stream back to the topic.

Apache Storm Apache Storm is a real-time computational framework. It consumes a stream

21. https://spring.io/projects/spring-cloud-stream visited 11.11.22

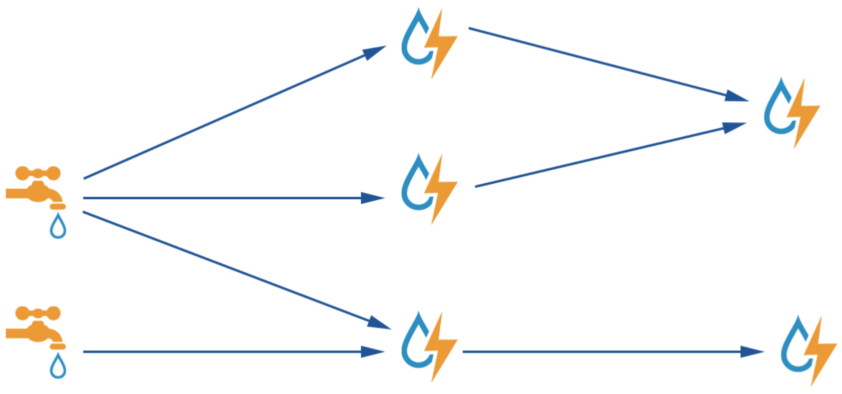


Figure 3: The topology from Apache Storm(Apache Foundation 2022b) Copyright © 2022 Apache Software Foundation.

of messages from a messaging broker or another data stream.(Jankowski, Pathirana, and Allen 2015, "Chapter 1. Introducing Storm ") The framework itself consists of Spouts and Bolts and streams. The Spouts are generating the stream from the messaging system it is connected to. The generated stream is a possible non-finite sequence of messages between Spouts and Bolts or Bolts and Bolts. The bolts are receiving messages from the stream and applying processing tasks on them. Bolts can again generate a stream or can persist data into a sink. The topology of Apache Storm can be seen in figure 3. Apache Storm does not have pre-defined processing tasks, like covered in 2.2.2. Although, there are multiple interfaces which can be utilized to process the data stream in Apache Storm. One of them is a SQL interface, which allows executing queries on the streamed data, another one is the Trident interface, which adds the common functionality of a stream processing frameworks, and also adds an exactly once guarantee.(Jankowski, Pathirana, and Allen 2015, "Chapter 9. Trident").

Apache Spark Apache Spark is a fast, reliable, and fault-tolerant distributed computing framework for large-scale data processing. (Garillot 2019, 48)

It can be connected to several existing streaming technologies like pub/sub brokers. But can also work on file streams and several data sinks, like databases or files. It stores the data in memory for caching, which results in a fast processing speed. It provides APIs for usual data processing methods 2.2.2 mentioned before. It also enables the interacting with the data as table. This is done with Spark SQL. Apache Spark offers batch processing as well as stream processing, and can be used from JVM based languages like Java, Scalar or Kotlin, but there are also APIs for R and Python. As the caching is happening in memory, it offers fault-tolerance by rebuilding the data by rebuilding it from the same data source. The streaming capability is added to Spark with the Spark Streaming or the Spark Structured Streaming plugins. The streaming plugin allows the usage of the processing methods from the batch processing core of Apache Spark, also on streams. The Spark Structured Streaming plugin adds further capabilities like the usage of Spark

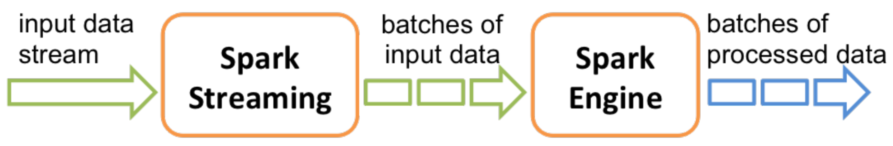


Figure 4: The topology of the Apache Spark Streaming(Apache Foundation 2022d) API

SQL and also adds further stream-specific features.
In contrast to the other stream processing frameworks, Spark does not yet support actual stream processing. It utilizes a method called microbatching2.2.4. This generates small batches which are defined by a fixed time interval(Garillot 2019, 45). In figure 4 it can be seen that the Spark Streaming plugin first generates small batches of the actual stream of messages, and they are then processed by the batching data processing engine of Apache Spark. For the streaming API, the fault-tolerance model, where data is rebuilt, is likely not available because the incoming data stream is not available anymore. Therefore, in the Streaming plugin, the incoming stream is replicated and distributed to multiple Spark instances(Zaharia et al. 2012, 3).(Garillot 2019, 11)

Apache Flink Apache Flink is a framework for stateful computations over unbounded and bounded data streams. Flink provides multiple APIs at different levels of abstraction and offers dedicated libraries for common use cases. (Apache Foundation 2022e)

Apache Flink 5 can be used to receive bounded, or unbounded streams of data from many common pub/sub brokers or file systems. In Apache Flink, every worker has its state, which is persisted using RocksDB[22]. It provides multiple APIs for the processing tasks. These are an SQL/Table API, which allows operating SQL queries on the streamed messages. An DataStream API, which provides the common functionality2.2.2 needed for stream data processing and an ProcessFunction API, for event-based tasks. Apache Flink offers an exactly once guarantee for the messages within the stream. It offers low latency, high throughput and can be scaled horizontal. Additional.(Apache Foundation 2022a)

2.2.4 Microbatching

With microbatching, processing task are not executed an any individual message, but on batches of messages. Those batches message for a predefined duration. The duration can be defined as required by the use case. It might be only milliseconds, but could also be several seconds. Microbatching can increase the throughput, but affects the latency, as the batching adds latency to all message.(Garillot 2019, 48)

22. https://rocksdb.org/ visited 29.11.22

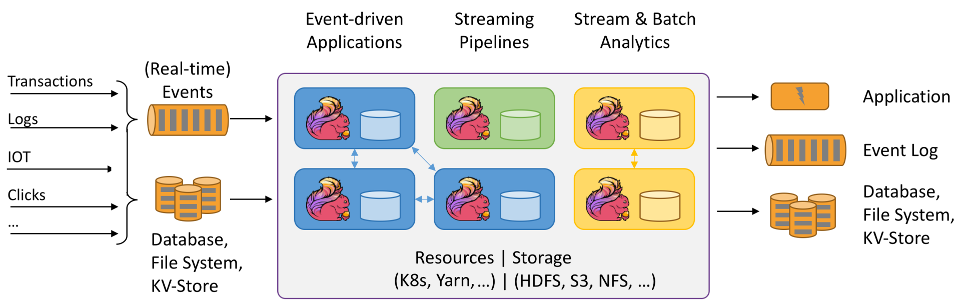


Figure 5: High-level overview of an Apache Flink pipeline(Apache Foundation 2022a) API

2.3 Related work

There are no papers regarding stream data processing with the usage of WebAssembly modules. Therefore, related research on the individual topics is considered. They are split into WebAssembly Container Runtimes, stream processing frameworks and data processing in the context of C-ITS.
For serverless WebAssembly, two surveys are analyzed. One work compares the performance of the Wasm modules to applications which are in Docker Containers (Long et al. 2021). Another work evaluation of WebAssembly based Container runtimes Gackstatter 2021.
As related work for stream processing frameworks. Papers, which compare the introduced stream processing frameworks2.2.3, are used.
For the last section, the data processing in the C-ITS context, a paper from Hugo, Morin, and Svantorp (2020), in which an approach for handling C-ITS messages that are sent from vehicles in a scalable way, has been analyzed.

2.4 Serverless WebAssembly

Gackstatter (2021) evaluated in his diploma thesis different WebAssembly runtimes, considering their performance in terms of cold start time of the runtimes and the execution time for different sets of workloads. In the thesis, the Apache OpenWhisk[23] serverless framework has been modified to support WebAssembly modules as well as Docker Containers. For the evaluation the framework has been executed on a Raspberry Pi Model 3B which is considered by the author as a fitting representation for IoT devices and a workstation with a server-grade 4 Core CPU.
The outcome of the work is, that in most cases, the cold start latency could be reduced by about 99.5%(Gackstatter 2021, 81) when executed on the Raspberry Pi and by about 94%(Gackstatter 2021, 56–57) on the server grade hardware, when the application is compiled to Wasm instead of being containerized. The throughput has been improved by a factor between 2.4 to 4.2x(Gackstatter 2021, 60). For CPU-bound workloads, the modules which are run and the

23. https://openwhisk.apache.org/

fastest of the compared runtimes is about 12%(Gackstatter 2021, 62) slower than the code running native within the Docker Container. However, the author claims that the benefits of the Wasm modules can make up the shortfall for many use cases, especially if the operations are not long-running and the startup of the application is also considered(Gackstatter 2021, 62).
The outcome of the diploma thesis from Gackstatter (2021) is, that Wasm is well suited for serverless applications, but still at an early-stage and important features were not yet implemented.

Long et al. (2021) demonstrates in the paper “A lightweight design for serverless Function-as-a-Service” advantages of lightweight high-level runtimes like WebAssembly in comparison with container-based systems like Docker or Firecracker. Long et al. (2021, 2) highlights the importance of a short start up time for Function-as-a-Service (FaaS) and that conventional container runtimes do not satisfy this need. Cloud providers use either modified container runtimes to reduce this issue, or have switched to Wasm-based solutions(Long et al. 2021, 2).
In the paper, benchmarks for several test cases have been run where the same application is executed within a Docker container, and compiled to WebAssembly and run in a Wasm runtime. The tests showed that the fastest Wasm runtime needs about 1/500 of the startup time of the Docker version. Read and write operations are between 50x and 150x faster, and for other synthetic scenarios the different were between 10% and 50% in favor of the Wasm runtimes(Long et al. 2021, 3–4).

2.5 Stream processing frameworks

Karimov et al. (2018) designed a benchmark for stream data processing frameworks. The benchmark contains various scenarios like window aggregations, window joins and others. The benchmark was applied to Apache Spark, Apache Storm and Apache Flink with the outcome, that all frameworks have their strengths and weaknesses. Apache Spark can handle scenarios where the load is not evenly distributed between the streams the best. Apache Flink is the best when a minimal latency and throughput is the main priority(Karimov et al. 2018, 1517).
Dongen and Van den Poel (2020) evaluated the latency, the throughput, CPU, and memory utilization from Apache Spark, Apache Flink and Kafka Streams. For this evaluation, there were four types of workloads defined Those workloads are a single burst of messages at the beginning of the stream, periodic bursts, a constant stream of messages with a minimal size and a constant stream of messages, where the number of messages gets increased over time. The workload for evaluating the latency showed that Apache Flink as well as Kafka Streams have significant lower latencies than Apache Spark, while Apache Flink showed also better results than Kafka Streams(Dongen and Van den Poel 2020, 1852). The workload with the increased number of messages showed that the microbatching approach of Apache Spark can significant increase the throughput, which the Structured Streaming API of Spark, it was possible to process 115,000 messages per second(Dongen and Van den Poel 2020, 1854). When the batching window is three seconds, the throughput is reduced to about 26,000 messages per second. As Apache Flink and Kafka Streams do not batch messages, their throughput was significant lower, with

about 18,500 messages per second for Kafka Streams and 26,000 for Apache Flink. For the test with the burst at the beginning, Apache Spark showed the best results, followed by Apache Flink(Dongen and Van den Poel 2020, 1855). The last showed that the micro batching approach of Apache Spark can reduce the load on a CPU and memory a lot, when it is compared to the actual streaming approaches. For the latency, there was again Apache Flink the fasted, followed by Kafka Streams(Dongen and Van den Poel 2020, 1856). The conclusion from Dongen and Van den Poel (2020, 1857) is, that the used framework should be chosen by the requirement. When a minimal latency is crucial, Apache Flink is the best choice, otherwise Apache Spark with its Structured Streaming API offers better values for throughput. Kafka Streams, on the other hand, is a valid alternative, as it does not require the setup of an own cluster, since it is part of Apache Kafka.

2.6 The reference system

2.6.1 Message generation

The messages which are processed in this pipeline are C-ITS Cooperated perception messages (CPM) which are defined within the C-ITS standard ETSI (2021). The CPM definition is still only a draft and there might be changes to the message format.
The CPMs are generated from two cameras which are placed at the intersection Weiserstrasse & Gabelsbergstrasse in Salzburg. They are recording the location of every street user with a 125 millisecond resolution in a local coordinate system. The detection of the street users is done with two cameras. Furthermore, additional information, such as if the object is a cyclist, a car, or a pedestrian, is recorded. Those detected objects are then sent to the roadside unit (RSU) which is also placed at the same intersection. In the RSU, the detected objects are then used to generate the actual CPM. If there are no objects detected, the CPM is still generated, but it will only contain metadata in such case. Before sending, the CPM gets wrapped into a message which contains additional metadata which is not part of the CPM. Those additional information is the timestamp when the message is sent, possible linked messages, tracing information and others.
The message is then sent to an AMPQ 1.0[24] broker, which is part of the data ingestion infrastructure from Salzburg Research. A message decoder service, which is also part of the data ingestion infrastructure, subscribes to the messaging queue and translates the C-ITS message into an internal Protocol Buffer-based messaging format and published the internal representation of the C-ITS message to the Apache Kafka instance which is the core component for the data streaming platform which will be evaluated against the WebAssembly based approaches.
For the evaluation, messages which passes the messaging decoder service have been recorded for a given timespan. The date, on which the test data was recorded was September 30th 2022 at 10:43. This time has been chosen because it is, according to Google Maps[25], a busy period. Those messages will be used for the comparison. The generated datasets consist of about 3,300

24. https://www.amqp.org/resources/specifications
25. https://www.google.at/maps/ visited 24.11.22

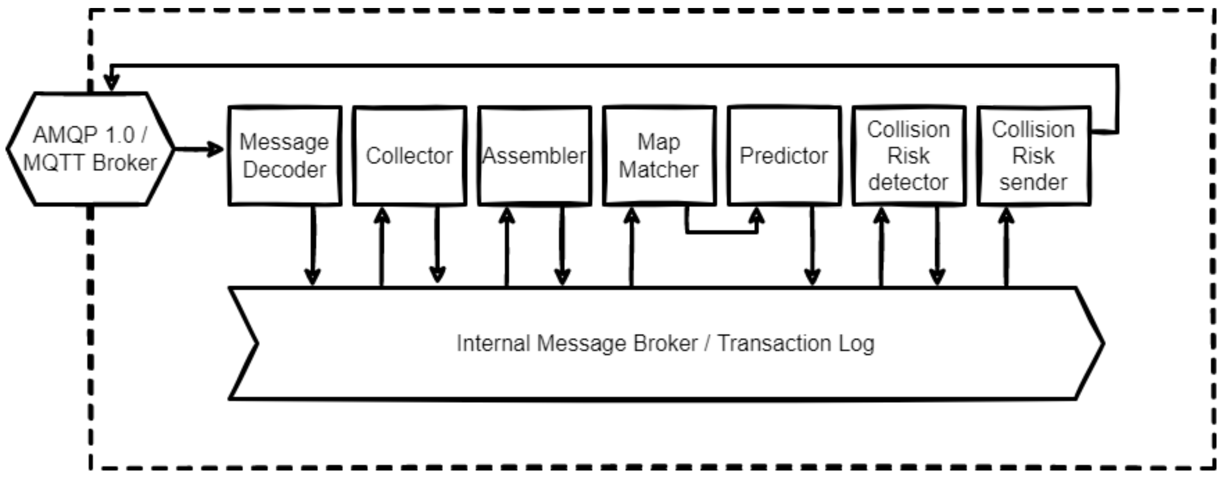


Figure 6: Overview over the entire platform, including only functional parts

messages, which will cause the pipeline to produce about 20,000 map matched trajectories. The same dataset will then be replayed twice as fast and four times as fast.

2.6.2 The entire stream data processing pipeline

For the comparison between the different stream data processing approaches, parts from an actual stream processing pipeline have been taken. The pipeline, which is used as reference, receives C-ITS Cooperative Perception Messages (CPM) (ETSI 2021, 31) which are recorded at an intersection, detects potential collisions and sends Decentralized Environmental Notification Messages (DENM), containing the collision risk, to affected C-ITS capable vehicles at the intersection. The whole pipeline consists of the following functional parts which are passed in sequential order, where the output of a service is the input from the next one.

C-ITS Message decoder The first part of the pipeline takes the C-ITS messages which are using ASN.1[26] as notation. These messages can either be encoded with the Unaligned Packed Encoding Rules(ITU-T 2021a)(UPER) or with the XML Encoding Rules(ITU-T 2021b)(XER). Once encoded, they get decoded into Protocol Buffers. It consumes the encoded messages from a message broker, translates them and sends them to a topic of a message broker. Protocol Buffers are used because the messages are not only used within the pipeline which is used within this use case, but also different services, which have been implemented in a way, that they cannot work with UPER encoded messages. In addition to this requirement, Protobuf allows fast serializing and deserializing of messages(Popić et al. 2016, 181), keeps the size small(Popić et al. 2016, 264) and allows for the usage of the message in multiple programming languages. This is because Protocol Buffers are supported by most programming languages.

26. https://www.itu.int/en/ITU-T/asn1/Pages/asn1_project.aspx visited on 22.11.22

Collector The second component in the pipeline is the collector. A collector takes incoming messages and extracts the relevant information like timestamps, coordinates and other information and puts those to an internal representation.This step is needed because some information from the C-ITS standardized messages needs additional information to handle properly, and to bring all the different message formats together into a single format. There may be multiple collectors, each of which handles only one kind of message type or message source. Depending on the input message, this service can be stateful or stateless and can be scaled horizontal.

Assembler The internal representation of the detected objects will then be consumed from the assembler. Within the assembler, objects are consumed, and a time-based sliding window function is applied to transform location-based messages into trajectory-based ones. This is for unique objects with a defined time frame. With every new consumed location, a new trajectory is generated and sent to the message broker.

Map Matcher The generated trajectories can origin from different sources, where the quality of the coordinates is unknown. In the Map Matcher, the trajectories are mapped to a given street graph (Quddus, Ochieng, and Noland 2007, 1) represented within the Lanelet2 format (Poggenhans et al. 2018). This is needed because GNSS date tends to be error-prone, especially in cities with higher buildings, which shadows the signal from the GNSS satellites. This step generates cleaned trajectories and adds knowledge about the semantics of the street, which is part of the street graph, to the trajectory.

Predictor The predictor uses the semantics of the street graph to predict possible paths. The different paths are then sorted by the probability they will be taken, and the most likely path will be elected as the prediction.

Collision detector In the collision detector, the predicted paths are used to create areas that mark a combination of spatial and time components. If multiple of those areas intersect, a collision risk is created.

Collision Risk sender The collision risk sender takes the collision risks, generate DENMs out of them and publishes them to a message broker. This service also handles duplicate which are caused by the risk detector, updates risks and provides information, if the collision risk changes.

For the evaluation only the Collector for CPMs, the Assembler and the Map Matcher are used since those three services already cover multiple use cases of stream processing pipelines such as stateless message transformation, session-based message aggregation and potentially slow algorithms applied to data. In the reference pipeline, the services are Java Spring Cloud Stream applications and Apache Kafka is used as data streaming platform. The CPMs are sent at 8Hz per sensor to the platform. Since every of the messages contains potentially up to 128 detected objects, which is defined by the ASN.1 representation provided by the C-ITS standard (ETSI 2021, 31), which are then passed to the assembler as unique messages. The assembler will then create another 128 messages for the map matcher. Therefore, the theoretical maximum number

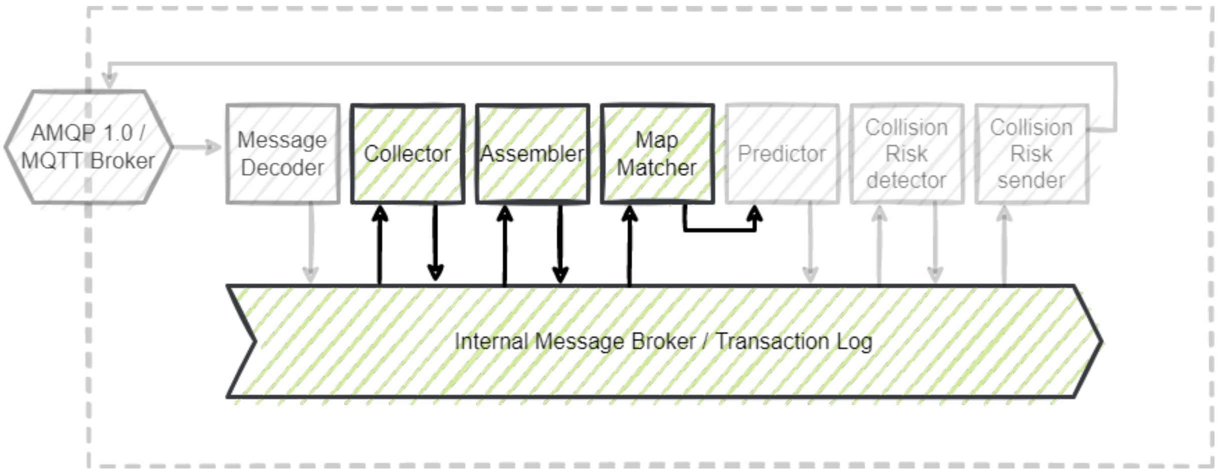


Figure 7: Selected parts to evaluate

of messages per second, the pipeline needs to be able to process for an intersection, which is equipped with a single camera, is 2,056. The data exchanged between different services are Protocol Buffer messages. The message format and schemas are identical for the compared implementations.

2.6.3 The reference implementation

The reference stream data processing platform is based on a microservice architecture using Apache Kafka for the internal communication. The different services consist out of Java Spring Cloud Streams applications 2.2.3. In comparison to the reviewed stream processing frameworks in2.2.3, the actual pipeline for the research project is not implemented as a single application, where the different processing steps are parts of the same service. The reason for this approach are frequent additions of new components, for example the collectors for different data sources, as well as different teams, working on different services. Because the entire reference implementation is deployed as an application for every processing step, the reference implementation, which is used for this evaluation, does it in the same way. The different services are running within containers, so it can be deployed on any cloud provider on virtual servers or can be managed by Kubernetes. From the chosen parts of the real-time pipeline, the assembler service uses Kafka Streams. The reason is that the aggregation functions from Kafka Streams can be utilized for this service.

Apache Kafka For the reference implementation, Apache Kafka is used. It is described in more details in 2.2.3.

Collector The collector service is a stateless application. As a result, it does not utilize the Kafka Streams Library. The first thing which happens in the collector is that the Protocol Buffer is deserialized to an internal representation of the CPM. Then the relevant information like timestamps and the positions of the detected objects as well as other

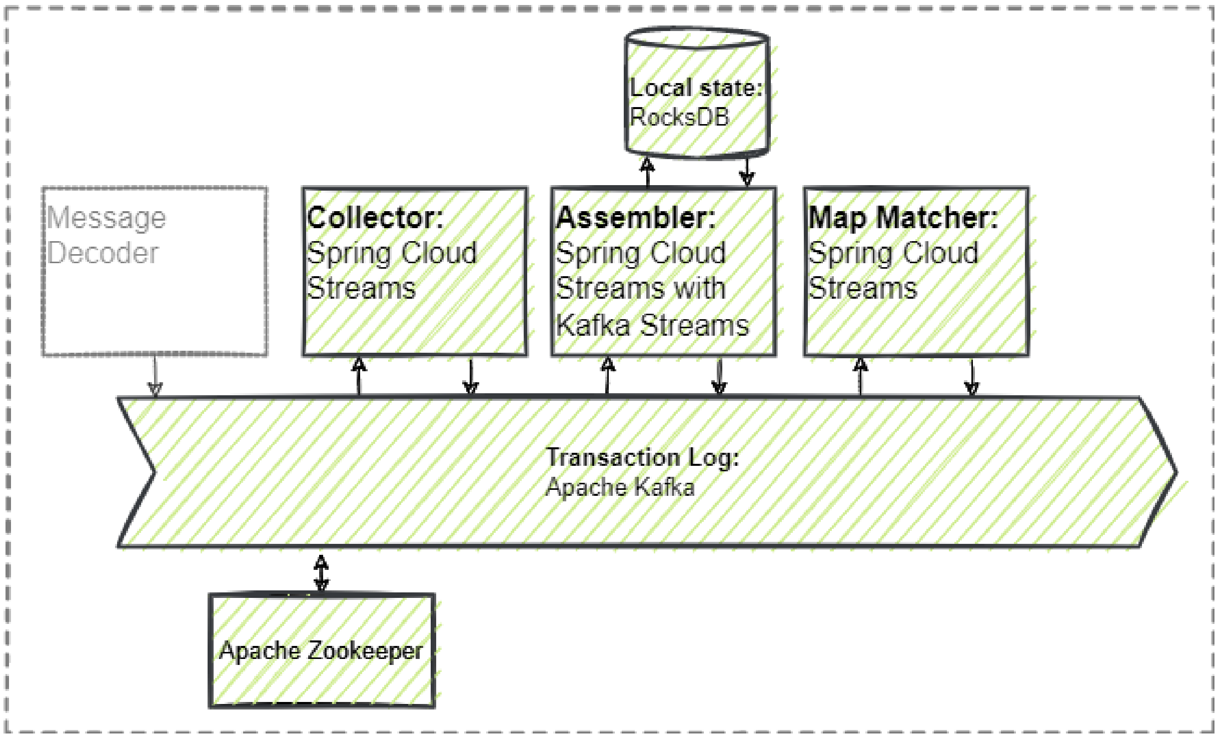


Figure 8: The reference platform which will be compared to Wasm approaches

properties of those are extracted. The mandatory information is the timestamp and the location. There might be situations in which the messages, containing the location are not in order, these situations are not handles within the collector. The handling is done within the assembler because to know if the order needs correction it needs to be compared with the other timestamps from the object. This is not possible when the service does not have a state. The locations are stored as relative coordinates to the camera which detected the objects. Those coordinates need to be converted from the local coordinate system to a geospatial one. In this case, to the WGS84 EPSG:4326 representation. The collector can generate multiple messages from a single input message. Therefore, the IDs from the single detected objects is used as key of the message for the Kafka producer. Because the C-ITS standard limits the IDs of the detected objects to 8bit, this leads to the issue that IDs are reused. This is also handled within the assembler, by closing the sliding window after a defined number of seconds. In the received messages, additional information are stored as number, or bit string encoded values [27]. For encoding, simple lookup tables or decoders are used. Once the data is extracted, the location-based messages are written to the Kafka log.

Assembler The assembler makes use of the Kafka Streams framework. Kafka Streams enable fault tolerance as well as parallel handling of messages by default. It comes with a local

27. https://www.oss.com/asn1/resources/asn1-made-simple/asn1-quick-reference/bitstring.html visited 11.11.22

state store, which is used for the window functions. For fault tolerance, every update of the state store is additional stored within a change log topic on the Kafka broker. This allows another Kafka Streams instance to take over if the service crashes for any reason. For parallelism, the library uses as many threads as there are partitions for the Kafka topic. Since the messages are partitioned using the message key, and there is always the same thread consuming from the same partition, race conditions are not possible. In the assembler, the location-based messages from the collector are consumed. If there is no data within the state store for the given key, a new trajectory will be created from the location. A trajectory can be seen as a list of locations with its corresponding additional properties encoded efficiently. If the key is already in the state store, the location is added to the trajectory. The timestamps of the trajectory are then corrected by sorting the newly added location into the already received ones. If a trajectory has not been updated within five seconds, it is assumed that the trajectory is concluded. Concluded trajectories are stored to a different Kafka topic, and the key which has been used to aggregate the locations can be reused. For the evaluation only the real-time trajectories are relevant, therefore finished trajectories will be dropped instead of stored within another topic.

Map matcher The map matcher consumes the aggregated trajectories in every iteration and tries to map the unique location to a given street graph. First, the application reads a geometry which represents the street graph from the file system. This geometry is stored in the Well-known Binary (WKB)[28] format. This format has been chosen because most programming languages have libraries that can parse it, and it is faster to parse than text-based representations of geometries like GeoJSON[28] or Well-known Text (WKT)[28]. In the following step, the actual map matching algorithm is applied. According to Singh et al. (2019, 65), map matching algorithms can be implemented on a pure geometrical, topological, or advanced way. Where the pure geometric implementations are based only on the geometries of the streets, while the topological ones also uses information from the roads. The advanced implementations are using algorithms using hidden Markov models (HMM) (Newson and Krumm 2009, 1) or others. The original pipeline uses the Graphium Neo4j[29] project, which utilizes an HMM based advanced map matching algorithm. In this implementation, only a very basic geometric algorithm is used, which uses the geometries from the street graph and moves every point from the trajectory to the nearest point of the graph geometry. This attempt has been chosen because it is less complex to implement and gets computational more expensive in every iteration of the trajectory, which is a typical scenario in for stream aggregations. Another reason for this simplified implementation is, that the map matching algorithms should be the same for the compared pipelines. Then resulting trajectories are then written to a Kafka log again, from where it could be processed further.

Scaling Scaling the reference implementation, implies the change of the configuration of the Apache Kafka messaging broker as well as the services. The reason for this is the handling of the consumer. Every consumer must have their partition in the message log at

28. https://www.ogc.org/standards/sfa visited 11.11.22
29. https://github.com/graphium-project/graphium-neo4j visited 11.11.22

the broker. If there are more consumers than partitions, for a given topic, the consumer without topic will wait until one of the other consumers stops working. This approach can be desired for redundancy reasons, but does not help when scaling out is the goal. Therefore, the number of partitions needs to be changed when services are scaled. Once the number of partitions is adjusted, additional instances of the services can be started. When the platform is run on a cluster with multiple nodes, there should be an Apache Kafka instance running on every node. Those are needed if data replication is required, and it also decreases the latency for the messages. When running on a single node, like it is the case within this evaluation, increasing the number of threads per service is sufficient to scale out the performance of the pipeline since the services are multithreaded. The Spring Cloud Streams implementations for the Kafka consumers are implemented multithreaded in a way, that the different threads are acting like they are different consumers within the same consumer group. Therefore, the number of threads configured in the Spring application needs to match the number of available partitions.

2.7 Possible WebAssembly based approaches

Until September 2022, there are only few options when it comes to server side WebAssembly frameworks. For the evaluation of the capability of WebAssembly for distributed stream processing pipelines, the following projects could be viable.

2.7.1 Fermyon Spin

Spin[30] is an open-source framework that supports running WebAssembly-based microservices. It uses the Wasmtime[31] WebAssembly runtime to execute the WebAssembly modules. The Spin framework is written in Rust and utilized Tokio[32] as asynchronous runtime (Fermyon Technologies 2022a). The Spin framework consists of a core component and is extended with so-called triggers and outbound capabilities. Triggers, listen for events, and once such an event happens, the trigger starts a Wasm module, which is linked to it, in its own WebAssembly instance. There are triggers for HTTP endpoint, message queues from a pub/sub brokers and timers available (Fermyon Technologies 2022b). The HTTP body or the received message from the queue is passed as payload to the microservice which is invoked. An outbound capability, on the other hand, is a set of functions which is injected into the WebAssembly module. Those functions can then be invoked, from inside the WebAssembly module. These functions are not executed within the WebAssembly module itself, but from the host runtime (Fermyon Technologies 2022a). Spin is taking advantage of parts of the WebAssembly component model, to enable this ability (Dice 2022). Behaviour, which is not part of the standardized WASI, such as making HTTP calls or publishing messages to a broker, is enabled this way. Currently, there are

30. https://www.fermyon.com/spin visited 11.11.22
31. https://wasmtime.dev/ visited 11.11.22
32. https://tokio.rs/ visited 11.11.22

only triggers for HTTP endpoints and Redis as pub/sub broker. The available outbound capabilities contain an HTTP client, a Redis interface and an experimental PostgreSQL client. Because the Spin framework is open source, additional outbound capabilities as well as triggers can be added if needed. Since the core of the Spin framework relies on an asynchronous runtime, the execution of the WebAssembly modules happens in an asynchronous manner, this way multiple services can be executed in parallel(Fermyon Technologies 2022a). With this approach, Spin provides concurrent execution. The Spin framework is part of the Fermyon platform[33], which enables to build a platform as a service for WebAssembly modules. It can be seen as an alternative to Kubernetes[34] with the difference, that it can only manage WebAssembly workloads(Butcher 2022a). The platform comes with a UI which can be used to deploy and manage the WebAssembly microservices, as well as monitoring and an ingress controller. It consists of only open-source components and runs under the Apache 2.0 license. There was an announcement, that there will also be a managed version of the Fermyon platform available. Because the Spin framework runs the WebAssembly modules only when needed, the platform scales as needed, while if there is no load, it scales to zero (Butcher 2022b). The cold start of the WebAssembly modules is usually in the single-digit millisecond range, or even shorter (Gackstatter 2021, 56). According to Fallin (2022) recent changes in the Wasmtime runtime made the startup time 400 times faster than before. These changes took place after the evaluation from Gackstatter (2021).

2.7.2 WasmEdge

The WasmEdge runtime has already been introduced before2.1.5. In contrast to other WebAssembly runtimes, it does extend the WASI standard with networking capabilities. This is done without using standardized approaches like the WebAssembly component model, therefore the WasmEdge runtime can run every WebAssembly module, but modules which are using the networking capabilities of WasmEdge can't be run in other runtimes (Second State 2022a). Like other WebAssembly runtimes, WasmEdge also has SDKs that can be used to run WebAssembly modules from other languages like Go, Rust, JavaScript, or others (Second State 2021b). Another feature is, that WasmEdge is supported by the crun[35] container runtime, which allows building a containerized application which only contains the runtime itself and the WebAssembly module. Such a container can be used like a traditional containerized application, but with less overhead (Second State 2021a).

2.7.3 WasmCloud

WasmCloud[36] is like Spin, an open-source framework which can execute WebAssembly modules. The modules are called actor in WasmCloud, referring to the actors in the actor model

33. https://www.fermyon.com/platform visited 11.11.22
34. https://kubernetes.io visited 11.11.22
35. https://github.com/containers/crun visited 11.11.22
36. https://wasmcloud.dev/ visited 11.11.22

(Agha 1986, 8) which is used in WasmCloud. In opposite to Spin, it uses Wasmer[37] as runtime and the actors are running not only when they are triggered. Like Spin, the WasmCloud host application provides the capabilities for the WebAssembly modules, which are not part of the WASI standard yet. Currently, this is not done with the WebAssembly component model, but according to Taylor (2022) this will change in the future. The host component is based on the open telecom platform (OTP) and is written in Elixir, while for the capabilities it uses Rust which can be executed from inside the Elixir host runtime (WasmCloud 2022b). The OTP framework is known to be battle tested and for its excellent scalability(Vinoski 2007, 2). For the communication between the actors, a Nats[38] broker is utilized. The default capabilities are Redis, PostgreSQL, HTTP, Nats, blob storage for file system and S3 and others. Unlike the implantation of the capabilities in the Spin framework, they are abstracted to so-called contracts. This means an application does only use a contract, without knowing how this capability is implemented. Such a contract could be for a key value store, which then could be implemented with Redis, Hashicorp Vault or any other key value store implementation. This allows a high level of abstraction and makes it possible to change the implementation of a capability without changing the application (WasmCloud 2022a). A focus from WasmCloud is security, which comes with the usage of WebAssembly. In comparison with other approaches, WasmCloud restricts the actors even more from the underlying operating system, allowing interaction only through the available capability providers. An example for this is getting the system time, which is not possible in WasmCloud as there is no capability for this available. WasmCloud is, like the Fermyon project, an installable platform. Modules, which are called actors, as well as their capabilities can be deployed from the build in web UI. It is also meant to be run directly on a server or inside Kubernetes.The WasmCloud platform is licensed under the Apache 2.0 license, but there is also a managed option available as developer preview called Cosmonic.

2.7.4 Spiderlightning

Within the writing, Deis Labs research introduced their Spiderlightning (Slight) project (Jiaxiao 2022). It is also an open-source project and should simplify building microservices. It provides a collection of predefined interfaces for common components for building microservices, such as key value store, message brokers, HTTP requests and others. The approach used by Slight resembles Fermyon Spin, in that the capabilities of WASI can be extended by a host runtime by utilizing the WebAssembly component model of WASI. This means that the host runtime provides interfaces that the WebAssembly module can use. Like the actors from the WasmCloud platform, the services are long-running processes, although with the addition, that they can also be stateful. In contrast to the other approaches, the host runtime from Slight is not shared with multiple services. Therefore, every WebAssembly module is executed by a different Slight instance. The currently implemented interfaces include, among others, Apache Kafka, Redis, and Azure Blob Storage.

37. https://wasmer.io/ visited 11.11.22
38. https://nats.io visited 11.11.22

3 System Overview

Within this section, the implementation of the WebAssembly-based approaches is described. From the possible frameworks2.7, which have been shown above, only Ferymon Spin, Wasm-Cload and Spiderlightning, also known as Slight has been chosen. The main reason for this, is that those frameworks adhere strictly to the WASI standard, or are in the process to implement it. This should enable some kind of compatibility of the WebAssembly modules between multiple vendors, and is not violating the capability-based security mechanisms from Wasm together with the WebAssembly system interface.

3.1 Fermyon Spin

The first evaluated WebAssembly implementation is the serverless approach, which uses the Spin framework. Spin has been chosen because the Redis pub/sub capabilities make it viable for the given use case. The different modules are written in Rust as it is one of the supported languages. The architecture of the Spin implementation is an event-driven microservice architecture like the reference implementation, but differs in several aspects. The first one is, that it is using Redis Pub/Sub instead of Apache Kafka. The next difference is, that the only service which runs permanently is the Spin host application and the Redis broker. The different parts of the actual processing pipeline are executed from the host component when needed. Another difference is, that the stateful component of the pipeline stored the state only within the distributed key value store.
Possible issues with an implementation with the Spin framework for stream processing are, that the default capabilities are rather limited. Since the number of capabilities is growing because the framework is still new, this issue might be only temporary. The age of the project, it is very young and fast evolving, breaking changes might happen more often. Another possible issue is, that although the Spin runtime executes the WebAssembly modules concurrently, the message broker might not. This can be solved with a Redis cluster instead of a single instance, but adds complexity. Another possible issue is the overhead for many short living WebAssembly instances and the start time of those virtual machines on every new message. This issue might also been resolved since with recent changes in the Wasmtime runtime, the startup time is now 400 times faster (Fallin 2022).

Redis[39] Redis is an in memory multipurpose data store. It is open source under a BSD license and can be used as key value store, database, publish and subscribe broker (Pub/Sub), as streaming engine and others. Redis itself stores data in memory but has multiple ways to persist the data, including database dumps or adding to an append-only log, but those mechanisms do not apply to the Pub/Sub component. Redis is single threaded only, and therefore should be used in the form of multiple Redis nodes within a Redis cluster if many concurrent writes and reads are necessary. Since adding complexity by operating a cluster of Redis nodes instead of a single one is not desirable, there are multiple feature complete drop-in replacements available which allow multithreading, one of them is

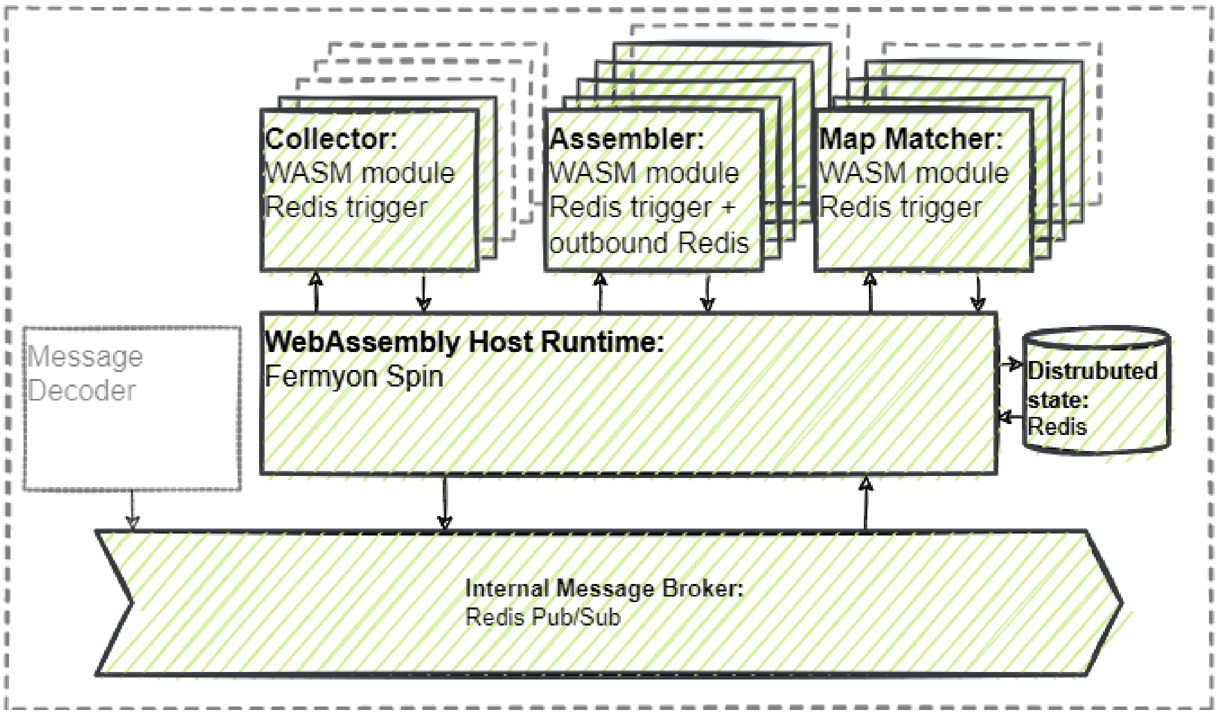


Figure 9: The serverless WebAssembly approach using Fermyon Spin

KeyDB[40] from Snap.inc. It will be used, if for the pipeline, Redis is identified as bottleneck. The Redis components which are used from the evaluated processing pipeline are the key value store as state for the assembler and Pub/Sub. Although there is also a build in stream engine called Redis Streams, it is not used, since Spin does not support it yet. Differences between Redis Streams and Redis Pub/Sub which might affect this Spin implementation are, that Redis Streams stores messages until they are evicted or deleted, while in Pub/Sub once messages are published they are gone. If there is no subscriber for a message, the message is lost. Since the use case of the processing pipeline is the generation of warnings in real time, the persistence of the messages is not crucial. Other differences are that Redis streams supports multiple groups for the clients. This means that multiple clients within the same group do not receive the same messages. Therefore, it can be used for horizontal scaling. In Redis Pub/Sub, every subscriber receives every message, so sharing work between workers is not possible. This is again not an issue when using Spin, since the host application receives the messages and starts asynchronously a worker for every one. This means, messages will be processed concurrently.

Collector The data collector is a simple Spin component with a Redis trigger and a Redis outbound capability. Because of the trigger, the host application will execute, the `on_message` function of the WebAssembly component, once a message is received for a defined messaging channel. In the example code 2 for the collector, it can be observed, that there is almost only the business logic within the module. The entire configuration happens

40. https://docs.keydb.dev/ visited 11.11.22

within the `spin.toml` file. The first thing that is always necessary, when the Redis interface from Spin is used, that the Redis address and channel is read from the environment variables, as in line 4. Then the message is deserialized from the received byte array in line 7. In this actual implementation, the deserializing is coupled with the extraction of the contained message. Since the used Protocol Buffer messages are language-agnostic, the messages, as well as the encoding, are identical to the reference implementation. The last part, in line 9 of the example code, is that the individual locations from the CPM are extracted, wrapped within the internal schema and published to the Redis broker.

```
 1  #[redis_component]
 2  fn on_message(message: Bytes) -> Result<()> {
 3
 4      let address = std::env::var(REDIS_ADDRESS_ENV)?;
 5      let channel = std::env::var(REDIS_CHANNEL_ENV)?;
 6
 7      let (_, msg) = proto_helper::get_cits_wrapper_from_bytes(message);
 8
 9      proto_helper::get_and_forwarde_location_container_from_wrapper(msg
            , &address, &channel);
10      Ok(())
11  }
```

Listing 2: The entry point of the collector of the Spin implementation

The publishing is done directly within the loop 3 over the locations, to keep the latency as low as possible. In line 9 some information from the metadata of the message, which is needed for the calculation of the absolute coordinates, is extracted. Then, in line 11, this information is used to generate a vector of the contained locations. Then, in line 13, it iterates over this vector. Within every iteration, in the lines 26 and 27, the message get encoded into the internal format, and published to the Redis broker.

```
 1  pub fn generate_and_send_locations(
 2      cpm: Cpm,
 3      wrapper: CitsMessageWrapper,
 4      address: &str,
 5      channel: &str,
 6  ) {
 7      let reference_timestamp = wrapper.timestamp;
 8      let source = format!("{}_{}", SOURCE_ENV, wrapper.provider);
 9      let (ref_lng, ref_lat, s_angle, e_angle) =
            get_reference_lng_lat_and_angles(&cpm);
10
11      let locations = get_locations(&cpm, ref_lng, ref_lat, s_angle,
            e_angle);
12
13      for (id, location, meta_tags) in locations.into_iter() {
14          let mut buffer = Vec::new();
15          let trajectory_id = id.to_string();
16
17          let lc = LocationsContainer {
18              trajectory_id,
19              timestamp: reference_timestamp,
            source: wrapper.uuid.clone(),
            owner: source.clone(),
            tags: meta_tags.clone(),
            locations: vec![location],
            trace: None,
        };
        prost::Message::encode(&lc, &mut buffer).expect("ERR: couldn't
            encode the message!");
        pubsub::publish(&address, &channel, buffer);
    }
}
```

Listing 3: The publishing of the individual locations

The details of the processing from the messages is handled identical to the reference implementation. With the same algorithms. Although those are implemented a bit different since in Rust the handling of data is a different from Java. Some examples are that data is immutable by default, objects passed to functions are moved and are freed once the function has finished, when passing references to functions they are not mutable by default and if they are passed explicit as mutable reference, it is not allowed to have another reference to the same data(Klabnik and Nichols 2022).These differences might lead to differences within the algorithm implementation.
After the data is extracted from the received messages, the newly generated message is encoded and published to the next channel.

Assembler The assembler uses, in addition to the Redis trigger and Redis outbound capabilities, a second Redis outbound capability. The second Redis connection is used as state for the windowing function. The first thing the assembler does after deserializing the received message is a lookup for the ID of the received location within its state. For this lookup, in line 8 from the example 4, the ID of the received location is extracted. Then, in line 10, this ID is used for a get command to the distributed Redis state. The `match` statement in line 11 matches the length of the returned value. Since the get command in line 10 is implemented in a way that it returns an empty vector if there is no result, the length of the payload can be used for pattern matching. If there was no trajectory in the cache, the `Option` for the cached track is set to `None`. In line 16, any other result than 0 will set the `Option` to `Some` with the cached track as value. In line 23 an ID is generated if there was no track in the state store. The actual assembling and merging happens in line 29. This function checks if the track from the state is `None` or `Some`, if there was no previous version of the track, a new one is generated, otherwise the newly received location is merged with the existing track.

```
1      #[redis_component]
2  fn on_message(message: Bytes) -> Result<()> {
3      //parsing envs
4      ...
5      //
6      let new_location: LocationsContainer = prost::Message::decode(
           message.as_ref()).unwrap();
7
```

```
 8      let trajectory_id = new_location.trajectory_id.clone();
 9
10      let payload: Vec<u8> = redis::get(&state_address, trajectory_id.
           as_ref()).map_err(|_| anyhow!("Error querying Redis"))?;
11      let existing_tracks_for_id: Option<Tracks> = match payload.len() {
12          0 => {
13              println!("track is not in cache");
14              None
15          }
16          _ => {
17              println!("track is in cache");
18              let track: Tracks = prost::Message::decode(payload.as_ref
                   ()).unwrap();
19              Some(track)
20          }
21      };
22
23      let id = match existing_tracks_for_id {
24          None => {
25              format!("{}{}",trajectory_id,  new_location.timestamp.
                   clone())
26          }
27          Some(_) => { "".to_string() }
28      };
29      let assembled_track = assemble_track_from_location(new_location,
           id, existing_tracks_for_id);
30
31      let mut buffer = Vec::new();
32      prost::Message::encode(&assembled_track, &mut buffer).expect("ERR:
            couldn't encode the message!");
33
34      redis::setex(&state_address, trajectory_id.as_ref(), buffer.
           as_slice(), 1)
35          .map_err(|_| anyhow!("Error executing Redis command"))?;
36      redis::publish(&address, &msg_channel, buffer.as_slice());
37
38
39      Ok(())
40  }
```

Listing 4: Main flow of the assembler of the Spin implementation

The data extraction, merging, and message generating part is, like the implementation in the collector, as close to the reference implementation as possible. So, points are extracted, sorted and the metadata is added. Once the message is generated, in line 34, it is written into the distributed Redis cache with the setex[41] command from Redis. This command sets a value in the key value store, that will be automatically removed after a given time.
Since Spin does execute the components concurrently, an implementation of the time

41. https://redis.io/commands/setex/ visited 30.11.22

based windowing as done here might lead to severe issues if there are multiple assembler instances processing the same trajectory ID at the same time. Possible issues are overwriting the newest version of the trajectory with an older one. With stream processing engines like Redis Streams or Apache Kafka Streams, such data races are not possible, like explained within the introduction for Apache Kafka2.2.3. A possible workaround for this issue is, to drop messages if the message from the cache is newer than the just assembled one. For the real-time use case this workaround might be viable, but it is not implemented for this evaluation, since it would lead to a reduced number of messages and will distort the result. The last task of the assembler, in line 36, is to publish the generated messages to the Redis Pub/Sub broker into a different channel.

Map matcher The map matcher is using the same WKB geometries and the same algorithms as the reference implementation. It is similar to the collector, but with the difference that it relies on static filed for the street geometries. Like in the collector and the assembler, the first thing this service does is, in line 6. In line 7 from the code example 5, trajectories of pedestrians are filtered. This means if the trajectory belongs to a pedestrian, the function returns early. Only if the trajectory belongs to a bike or a car, the geometries are loaded. This is done in line 11 and 12 with the get command from the Redis key value store. Compared to the reference implementation, the use of static files in this serverless implementation leads to the issue that on every new message, the static file has to be required from the key value store, be deserialized and converted to a geometry. In the reference implementation, this happens only when the service starts for the first time. The conversion from the WKB byte array to the geometry happens in line 15 and 16. The `match_track_on_centerline` function iterates over every location within the trajectory, finds the closest point on the street geometry and moves the location from the trajectory to the found point. The result is a vector of point geometries, which is then in the final processing step at line 19 converted to a representation of the map matching result. Like with all other services, the last parts of the map matcher, in line 22 and 23, are to encode and publish the result to the broker.

```
1       #[redis_component]
2   fn on_message(message: Bytes) -> Result<()> {
3       //parsing envs
4       ...
5       //
6       let track: Tracks = prost::Message::decode(message.as_ref()).
           unwrap();
7       if is_pedestrian(&track) {
8           println!("skipping, pedestrian!");
9           Ok(())
10      } else {
11          let bike_centerline_vec: Vec<u8> = redis::get(&address, "
                graphs:bike").map_err(|_| anyhow!("Error querying Redis"))
                ?;
12          let car_centerline_vec: Vec<u8> = redis::get(&address, "graphs
                :car").map_err(|_| anyhow!("Error querying Redis"))?;
13
14
```

```
        let bike_centerline: MultiLineString = hd_graph::
            centerline_from_wkb_vec(bike_centerline_vec);
        let car_centerline: MultiLineString = hd_graph::
            centerline_from_wkb_vec(car_centerline_vec);

        let matched_points = matcher::match_track_on_centerline(&track
            , &bike_centerline, &car_centerline);
        let matched_track = track_builder::build_matched_track(&track,
             matched_points, new_trace_in_entry);

        let mut buffer = Vec::new();
        prost::Message::encode(&matched_track, &mut buffer).expect("
            Err: failed to encode message");
        redis::publish(&address, &channel, buffer.as_slice());

        Ok(())
    }

}
}
```

Listing 5: Main flow of the map matcher of the Spin implementation

Configuration The configuration from the service is decoupled from the actual code, in the Spin framework. This is done inside the `spin.toml` file 6. Within this configuration file, the different capabilities from all services are set. Line 1 to 4 are only metadata. In line 5 the type of the component is defined, in this case it is a `trigger` for Redis. Line 8 is the version of the application itself. Because the project consists out or more than just one WebAssembly component, there are multiple `[[component]]` blocks. Those represent the different components. The environment, like in line 9, is used for environment variables. The id in the line 10 is again metadata, but for the component and the source, defined in line 11, is the relative path to the WebAssembly module from the `spin.toml`. In line 12 the config for the trigger begins, in this case it is always only the channel of the Redis pub/sub broker. In line 18, with the files option, files, or directories can be passed to the WebAssembly module. In this case, only these files are accessible for the Wasm component.

```
1  spin_version = "1"
2  authors = ["Stefan Eisl <stefan.eisl@salzburgresearch.at>"]
3  description = "assembles trajectories from locations"
4  name = "c-its-processing-pipeline"
5  trigger = { type = "redis", address = "redis://localhost:6379" }
6  version = "0.1.0"
7
8  [[component]]
9  environment = { REDIS_ADDRESS = "redis://127.0.0.1:6379",
       STATE_STORE_ADDR = "redis://127.0.0.1:6379", REDIS_CHANNEL = "
       trajectories"}
10 id = "assembler"
11 source = "components/assembler-2.6.wasm"
12 [component.trigger]
```

```
channel = "locations"

[[component]]
environment = { REDIS_ADDRESS = "redis://127.0.0.1:6379",
    REDIS_CHANNEL = "final-msg", GRAPH_CHANNEL = "graph", GRAPHNAME = "
    Salzburg_Lanelet2_2021_09_24" }
id = "predicter"
source = "components/predicter-1.0.wasm"
files = ["static/*"]

[component.trigger]
channel = "trajectories"
...
```

Listing 6: Example for the config of the Spin implementation

Scaling Scaling is handled by the way Spin works automatically, When it runs on a single server. Every message starts an own instance of the service, many messages will lead to many services while if there are no messages, there are no services running. If the pipeline runs on multiple servers, this behavior is not available at the moment. When running on multiple server nodes, the only way to share work between them is to split the WebAssembly modules between them. For example, one server is running only the collector WebAssembly module, while another one runs the assembler and a third the map matcher. This way, it is ensured, that messages are not received, and processed, by multiple instances. This is necessary because Spin does not support Redis Streams, or another proper streaming engine yet. Simply deploying multiple spin instances, would mean that every instance receives every message.

3.2 WasmCloud

In the second evaluated WebAssembly implementation, the WasmCloud framework is utilized. WasmCloud has been selected because the utilized Nats Jetstream streaming engine has similar capabilities as Apache Kafka, and the provided capabilities should enable an implementation of the pipeline. The modules are, like in the Spin implementation, written in Rust. Therefore, many parts can be reused from the other implementation. Reusing the entire WebAssembly file is not possible, since the different framework are using different interfaces for the extended the WASI capabilities. Another difference is, that WasmCloud is more restrictive than Spin. While Spin has included capability to work with the file system, WasmCloud has not. It is also not possible to call the current system time from WebAssembly modules executed from WasmCloud. This is a security measure which prevents side channel timing attacks like Spectre(Cloudflare 2022). This limitation makes the proposed tracing impossible at module level. Fortunately, the overall latency of the pipeline can still be measured because there is an outbound timestamp added when the message is produced from the testbed as well when the result of the map matching service is received at the testbed. WasmCloud uses Nats as broker for the messaging between the components, which has a comparable feature set to Apache Kafka. The architecture is almost identical to the one in the Spin framework, with the exception that the services

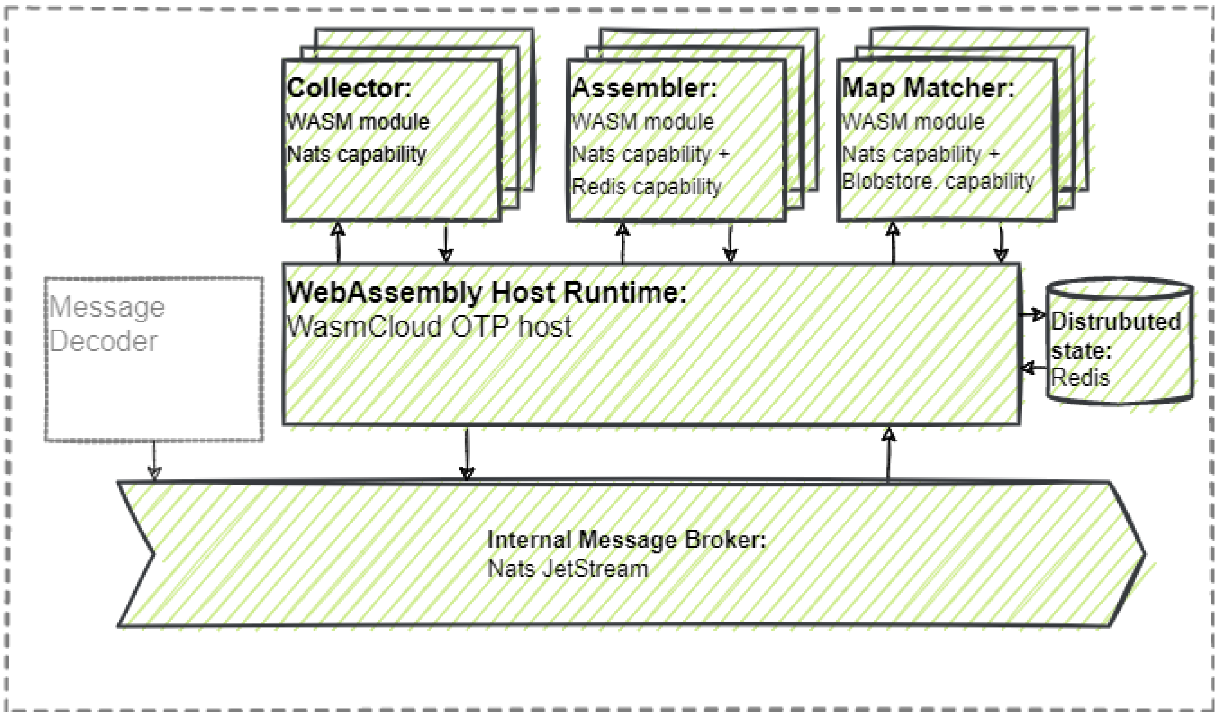


Figure 10: The WebAssembly approach using WasmCloud

are long-running processes and not only started if a message is received and the number of instances per actor is a defined number and not depending on the number of incoming messages. The different services are like in the Spin implementation stateless, and therefore the assembler service uses a distributed key value store for its state. For the state, a Redis instance is used because WasmCloud offers a capability for Redis. Configuring WasmCloud can be done via the web interface. Once WasmCloud is started, actors can be started and scaled from the web UI. The implementations for the capability providers can be selected. As example, for the blobstore capability, the provider for the file system or S3 can be loaded. Since the WebAssembly module only implements the interface of the capability, the actual implementations are exchangeable, as long as the interface is compatible.

Nats is a connective technology which acts as a message broker and also offers a streaming engine like Redis Streams or Apache Kafka called JetStream. Nats is designed to be lightweight and can be connected to other Nats nodes. An advertised use case is to run it on devices at the edge of the network(Nats.io 2021), which are then connected to other Nats service running in a cloud environment. When using multiple Nats nodes, messages can be routed via different paths to each other, which leads to a self-healing mesh which can still route messages even if individual nodes are not reachable. In addition, it is compatible with the MQTT[42] standard, which has been shown by Hugo, Morin, and Svantorp (2020, 375) to be a viable approach for the communication between connected vehicles and the processing infrastructure. The built-in streaming engine allows similar

42. https://mqtt.org/ visited 11.11.22

grouping for messages like Apache Kafka, which enables an exactly once message delivery and horizontal scaling when multiple subscribers are used. Because the messages are persisted when Nats JetStream is used, messages can also be replayed again.

Collector The data collector is realized as an actor which implements the message subscriber interface from WasmCloud. This way the actor subscribes on startup to a given subject of the stream. Because of the security-based restrictions, there are some changes, like the implementation for the tracing object, here the timestamps are replaced by zeros as there is no way to read the current system time. This is not shown in the example code of 7.

```
 1 #[async_trait]
 2 impl MessageSubscriber for CpmCollectorActor {
 3     async fn handle_message(&self, ctx: &Context, msg: &SubMessage) ->
           RpcResult<()> {
 4         let (_, message) = proto_helper::get_cits_wrapper_from_bytes(
               msg.body.as_slice());
 5         match &message.payload {
 6             Payload::SerializedMessage(m) => {
 7                 match prost::Message::decode(m.payload.as_slice());
 8                     Ok(data) => {
 9                     let lcs = generate_and_send_locations(data, &
                           message);
10                         let provider = MessagingSender::new();
11                         for (id, location, meta_tags, should_send) in
                               lcs.into_iter() {
12                           let mut buffer: Vec<u8> = vec![];
13                           if should_send.clone() {
14                             let trajectory_id = id.to_string();
15                                 let lc = LocationsContainer { ... };
16                                 prost::Message::encode(&lc, &mut
                                       buffer).expect("Err: failed to
                                       encode message");
17                                 provider.publish(ctx,&PubMessage {
18                                             ...
19                                         }).await
20                           ...
21                           }}}}}}
22  }
```

Listing 7: Main flow of the collector of the WasmCloud implementation

Additionally, the code needed to be rewritten to an asynchronous implementation, since the message subscriber interfaces are defined as such. Like in the Spin applications, the code only contains the business logic. The entire config happens within a web UI, or via a config file. The first part, in line 8, is the deserializing of the Protocol Buffer message and the extraction of the actual message, this has been reused from the Spin implementation, since both are written in Rust. In line 9, it is checked if there is a message within the wrapper. If there is a proper CPM contained, it gets decoded in line 12. In line 15, a slightly the locations are extracted from the message, this part is again mostly taken from the Spin implementation. Subsequently, in the lines 17 to 27, the application iterates over the locations and publishes those.

Assembler The assembler is also using the message subscriber interface like the data collector. Like the Spin implementation, the distributed state is also implemented via Redis as an external key value store. Furthermore, like with the collector, the logic of the implementation has been taken from the Spin implementation with minor changes because of the asynchronous implementation of the message subscriber. Redis has been chosen, since the support for the Nats-based version has been removed after the version 0.18 of WasmCloud. Another difference to the implementation is the way, trajectories are stored in the distributed key value store. The capability provider for the key value store is implemented in a way that it only accepts valid UTF-8[43] strings as values. This leads to the issue, that a serialized Protocol Buffer does not guarantee that the result is a valid UTF-8 string. For this reason, the resulting trajectory from the assembler is serialized to a JSON string before it is sent to the Redis. Compared to the binary representation, the use of a string representation for the messages is slow and might lead to a bottleneck. From Hugo, Morin, and Svantorp (2020, 375), the JSON parsing has been shown to be the most time-consuming part of the evaluation. The code example left some details, as they are similar to the already shown Spin version.

In line 6 from the code example 8, the previous version of the trajectory is loaded. Then later at line 11 the ID is set. The merging and assembling of the trajectories with the newly received locations is done in line 14. Then in line 16 and 17, the trajectory gets serialized into a JSON string, as described above, followed by the set command for the Redis key value store in line 19. The last task of the assembler is the publishing of the actual result message to the broker. This is still using the binary represented Protocol Buffer.

```
 1    #[async_trait]
 2  impl MessageSubscriber for AssemblerActor {
 3
 4      async fn handle_message(&self, ctx: &Context, msg: &SubMessage) ->
           RpcResult<()> {
 5          // ...
 6          let kv_content = KeyValueSender::new().get(ctx, trajectory_id.
              as_str()).await?;
 7
 8          let existing_tracks_for_id: Option<Tracks> = match &kv_content
              .exists {
 9            // same as in Spin
10          };
11          let id = match &existing_tracks_for_id {
12            // same as in Spin
13          };
14          let assembled_track = assemble_track_from_location(
              new_location, id, existing_tracks_for_id,
              new_trace_in_entry);
15          let internal_track_for_cache = trajectory::
              tracks_to_tracks_internal(&assembled_track);
16
```

43. https://www.ietf.org/rfc/rfc3629.txt visited 11.11.22

```
        let track_for_cache = serde_json::to_string(&
           internal_track_for_cache).unwrap();
        // encoding as in Spin
        let set_request = SetRequest {key: trajectory_id,
                value: track_for_cache,
                expires: 1
        };

        KeyValueSender::new()
            .set(ctx, &set_request).await?;

        MessagingSender::new()
        .publish(ctx, &PubMessage {
            subject: "cits.trajectories".to_string(),
            reply_to: None,
            body: buffer.to_owned()
        }).await?;

        Ok(())
    }

}
```

Listing 8: Main flow of the assembler of the WasmCloud implementation

Map matcher The map matcher implements, like the other WasmCloud actors, the message subscriber interface and uses the logic from the equivalent Spin module. Similar to the other actors, there were minor changes necessary. Those were the tracing objects which writes zeros instead of actual timestamps to the trace, the asynchronous behavior and the handling for the geometries which are read from the file system. Since the WebAssembly modules do not have direct access to the file system, the blobstore capability provider interface is used. This interface can be used for Simple Storage Service[44] (S3) or S3 compatible object storage instances or for the file system. This interface allows writing to a directory which is defined as configuration from the WasmCloud host. Similar as in the Spin implementation, in which the geometry is loaded from Redis for every message, the geometry will be read on every incoming message from the file system.
In line 7 from the code example 9 the blobstore connection is initialized, then in line 9, the geometry is loaded from the blobstore API. In line 24, the same function as in the Spin implementation is used for parsing the byte array from the WKB file and converting it to a geometry object. The functions for the actual map matching in line 29 and 31 are identical to the ones from Spin. Once again, the message is published to the Nats broker when the trajectory has been mapped to the geometry. Finaly the message gets encoded in line 38 and published in 39.

```
1       #[async_trait]
2 impl MessageSubscriber for MapMatcherActor {
3
```

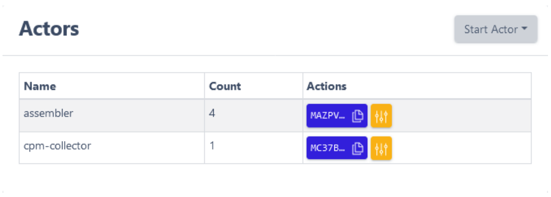
44. https://aws.amazon.com/s3/ visited 12.11.22

```
    async fn handle_message(&self, ctx: &Context, msg: &SubMessage) -> RpcResult<()> {

        // ...
        let blobstore = BlobstoreSender::new();

        let bike_graph_binary = match blobstore
            .get_object(
                ctx,
                &GetObjectRequest {
                    ...
                },
            )
            .await?
            .initial_chunk {
            None => {
                error!("MAPMATCHER: can't read graph from blob storage");
                vec![]
            }
            Some(graph) => { graph.bytes}
        };
        let bike_centerline = hd_graph::centerline_from_wkb(bike_graph_binary);

        // the same as above but for the car geometry
        //...

        let matched_points =
            matcher::match_track_on_centerline(&track, &bike_centerline, &car_centerline);
        let matched_track =
            track_builder::build_matched_track(&track, matched_points, new_trace_in_entry);
        match &matched_track.trace{
            None => { warn!("MAPMATCHER: no traces attatched")}
            Some(tr) => { info!("MAPMATCHER: length of trace list: {}", tr.trace_list.len())}
        }
        let mut buffer = Vec::new();
        prost::Message::encode(&matched_track, &mut buffer).expect("Err: failed to encode message");
        MessagingSender::new()
            .publish(
                ctx,
                &PubMessage {
                    ...
                },
            )
            .await?;
        Ok(())
    }
```

```
}
```

Listing 9: Main flow of the map matcher of the WasmCloud implementation

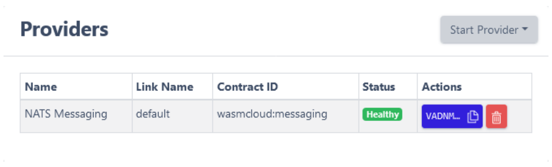
Configuration Like with the Spin implementation, the code is decoupled from the configuration of the interfaces. Since WasmCloud, offers multiple implementation for their interfaces, the one, which should be used, has also to be configured. This can be done within the web UI 11. The different implementations for the interfaces are called capability providers and needs to be stared if they should be utilized 11b. Then the actors can be started, either with uploading the Wasm binary from the file system or from a registry 11a. When the actor is started, the number of threads needs to be defined. Although this can be adjusted at runtime. At this state, the actors are already running, but not configured. At last, the actors need to be linked to the required capability provider 11d. Within the linking, also the configuration like, security settings or topic of the pub/sub broker is set 11c. Once this configuration is done, the actors are ready.



Actors

Name	Count	Actions
assembler	4	MAZPV...
cpm-collector	1	MC37B...

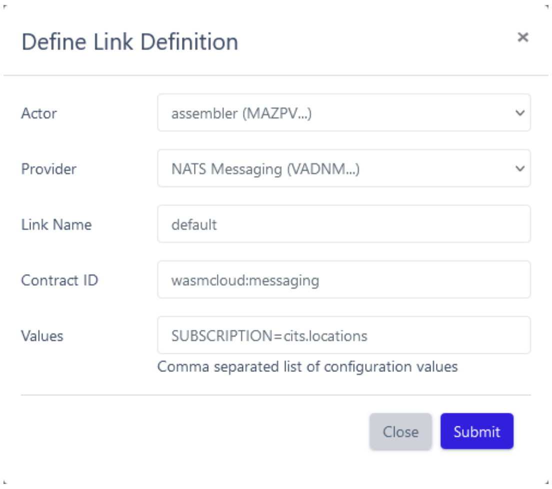
(a) Adding an actor



Providers

Name	Link Name	Contract ID	Status	Actions
NATS Messaging	default	wasmcloud:messaging	Healthy	VADNM...

(b) adding a provider

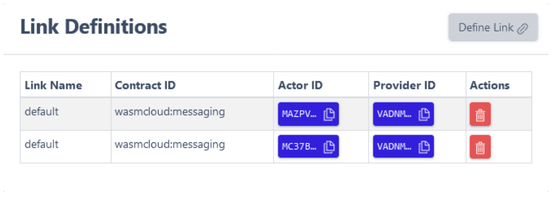


Define Link Definition

Actor	assembler (MAZPV...)
Provider	NATS Messaging (VADNM...)
Link Name	default
Contract ID	wasmcloud:messaging
Values	SUBSCRIPTION=cits.locations

Comma separated list of configuration values

(c) linking actors to a provider



Link Definitions

Link Name	Contract ID	Actor ID	Provider ID	Actions
default	wasmcloud:messaging	MAZPV...	VADNM...	
default	wasmcloud:messaging	MC37B...	VADNM...	

(d) Linked actors

Figure 11: The web UI for configuring, and linking WasmCloud actors

Scaling Scaling the WasmCloud implementation can be done, by simply using multiple WasmCloud instances on different servers. Once the Nats instance of the servers needs to be connected, Nats takes care over the routing of the messages to the proper WasmCloud service. Since WasmCloud is using Nats Jetstream, messages with the same key, will always be sent the same instance, if the deterministic token partitioning[45] strategy is used. This is crucial, because of the state which is hold in a Redis instance for the assembler service. When Wasmcloud is running on a single server, the number of instances per actor can be adjusted within the web UI. An automated scaling is not available.

3.3 Spiderlightning

The final evaluated WebAssembly approach for the stream data processing utilizes Spiderlightning. Spiderlightning, also called Slight, has been chosen, because of its Apache Kafka capabilities. This allows to use the pipeline as drop-in replacement for the reference pipeline. Another reason for choosing Slight is that it follows a different concept than the other approaches, by using an own runtime for every service and not a shared one for all services. Slight can be seen as a loose collection of interfaces which can be used for building web services. Therefore, the pipeline could be built from various brokers as the core component for the pipeline. Since Apache Kafka is already used in the reference implementation, it will also be used for this one. Like the WasmCloud approach, the services are also running if there are no messages to process, this eliminates the startup time of the modules. Unlike the other evaluated Wasm approaches, the services can have their state. This eliminates the possible bottleneck of fetching data on every incoming message. There is no direct access to the file system when using Spiderlightning, although there is a capability on the roadmap(Deis Labs Research 2022) which promises a similar approach as WasmCloud, which uses the interface for the object store. Unlike Spin, this approach does not scale the number of instances as needed. When using Apache Kafka, it is important, that the number of instances of the service matches the number of partitions, or is higher. This enables concurrency and provides the safety against data races.

Collector The data collector uses almost the same implementation as the one from the Spin approach. The only differences between them are the interfaces for the messaging broker. It uses the Pub/Sub interface, which is implemented for Apache Kafka. Furthermore, unlike the already described WebAssembly implementations, the entry point of the service is not a function that is called, like the `on_message` from Spin. This means, that the subscribing to the broker, has to be implemented to the application. This is done by connecting to the broker via the given interface and then waiting for incoming messages inside an infinite loop.
In the code example 10, in line 2 the subscriber of the pub/sub interface calls `open` which gets the configuration from the `slightfile.toml`, in which the used resources for the broker, in this case Apache Kafka, is stored. In line 3 the same thing happens, but for the message producer. Line 4 connects to the broker, using the credentials from the configuration. The infinite loop, mentioned earlier, starts at line 7. Within this loop, at line 8 the next message from the broker is received, from which the content is extracted in line 9, with the same code as in the Spin implementation. In line 10, a little modified version of

45. https://docs.nats.io/nats-concepts/subject_mapping#deterministic-subject-token-partitioning visited 13.11.22

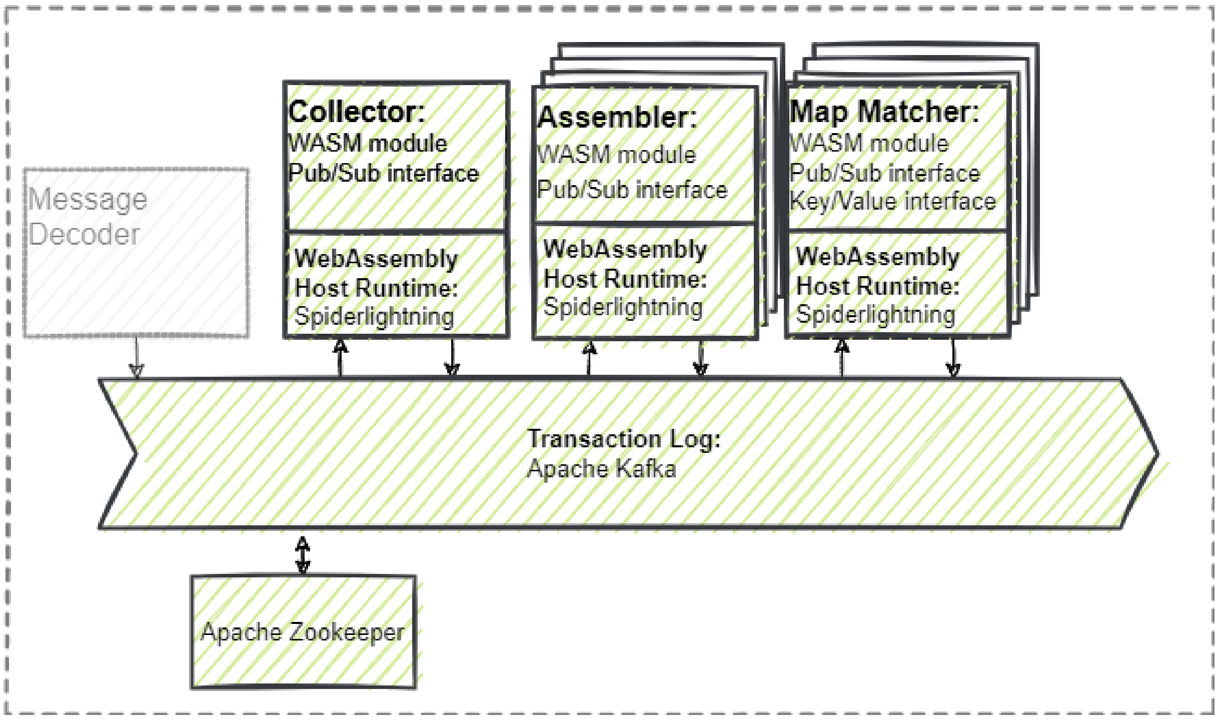


Figure 12: The WebAssembly approach using Slight

the Spin code, which is shown in 3, is used. The difference is, that the connection for the publisher is passed into this function.

```rust
fn main() -> Result<()> {
    let sub = Sub::open("my-pubsub")?;
    let pubs: Pub = Pub::open("my-pubsub")?;
    sub.subscribe(IN_TOPIC)?;
            println!("started collector");

    loop {
        let message = sub.receive()?;
        let (uuid, msg) = proto_helper::get_cits_wrapper_from_bytes(
            message.as_slice());
        proto_helper::get_location_container_from_wrapper(msg, &pubs,
            OUT_TOPIC, new_trace_in_entry);
    }
    Ok(())
}
```

Listing 10: Main flow of the collector of the Slight implementation

Assembler The assembler reuses like the collector most of the code from the Spin implementation. The main difference is, that it doesn't use an external state, since it is a stateful application. Instead of the Redis key value store, the assembler utilizes an in memory cache, which evicts entries after a given time. Since WebAssembly can only address 32 bit of memory, it is important, that this cache is cleared frequently to avoid runtime er-

rors. Furthermore, since the cache for the currently active trajectories, is not distributed, a crash of this service, would lead to the behavior, that new points, which would have been merged with existing trajectories, will be handled as new ones. For this specific use case, this is no issue, as a split trajectory into two parts won't cause issues. The message handling is implemented like in the collector. Since the assembler will receive significant more messages as the collector, running multiple instances might be needed.
In the code example 11, the loading of the configurations of the interfaces, in the lines 2 and 3 is identical to the collector. Then, in line 5, an in memory cache is initialized with a time to live of one second. Line 7 the connection to the broker is established for the consumer. Inside the infinite loop, the trajectory gets received in line 9 and deserialized and decoded in line 10. the lines 11 to 25 are almost identical to the Spin implementation, with the difference, that instead of the get command from Redis, the in memory cache is used. The assembling happens inside the function, that is called at line 21, and the trajectory gets inserted to the cache at line 26. The rest of the code is encoding and publishing the message.

```
1  fn main() -> Result<()> {
2      let sub = Sub::open("my-pubsub")?;
3      let pubs: Pub = Pub::open("my-pubsub")?;
4      let time_to_live = ::std::time::Duration::from_secs(1);
5      let mut cache = LruCache::<String, Tracks>::with_expiry_duration(
           time_to_live);
6
7      sub.subscribe(IN_TOPIC)?;
8      loop {
9          let message = sub.receive()?;
10         let new_location: LocationsContainer = prost::Message::decode(
               message.as_slice()).unwrap();
11         let trajectory_id = new_location.trajectory_id.clone();
12
13         let existing_tracks_for_id = cache.get(&trajectory_id);
14         let id = match existing_tracks_for_id {
15             None => {
16                 // if the track does not exist generate a ID
17                 format!("{}{}", trajectory_id, new_location.timestamp.
                       clone())
18             }
19             Some(_) => "".to_string(),
20         };
21         let assembled_track = assemble_track_from_location(
22             new_location,
23             id,
24             existing_tracks_for_id,
25         );
26         cache.insert(trajectory_id.clone(), assembled_track.clone());
27         let mut buffer = Vec::new();
28         prost::Message::encode(&assembled_track, &mut buffer).expect("
               Err: failed to encode message");
29
30         pubs.publish(buffer.as_slice(), OUT_TOPIC );
```

```
    }
    Ok(())
}
```

Listing 11: Main flow of the assembler of the Slight implementation

Map matcher The map matcher is, like the other parts of this implementation, almost identical to the Spin implementation. Like the assembler, it has a local state. The state is used to store the street geometry, which eliminates the reading and deserializing of the well-known binary on every received message. Unfortunately, Spiderlightning does not grant access to the file system and there are no capabilities for accessing it implemented yet. The geometries are stored in a key value store, like in the Spin implementation, where they will be read at the initial startup of the service. Once the state is set up, and the connection to the broker is established, the service waits for messages within an infinite loop. Received messages are handled as in the Spin implementation. Since the map matching service represents a possible slow algorithm, multiple instances of this service will run concurrently.
The code snipped 12 shows, that the first thing that happens, in the lines 2 to 4, is the initializing of the interfaces with the given configurations. In this implementation, there is also a config for the key value store read. the reading of the street geometry, from the key value store, in the lines 5 and 6. This is followed by the converting the byte arrays to geometry objects in the lines 7 and 8. Line 9 connects to the broker, and the logic within the infinite loop from line 10 onwards, reassembles the map matcher implementation from Spin again.

```
1  fn main() -> Result<()> {
2      let kv = Kv::open("graphs")?;
3      let ps = Sub::open("my-pubsub")?;
4      let pubs: Pub = Pub::open("my-pubsub")?;
5      let bike = kv.get("bike")?;
6      let car = kv.get("car")?;
7      let bike_centerline: MultiLineString = hd_graph::
           centerline_from_wkb(bike);
8      let car_centerline: MultiLineString = hd_graph::
           centerline_from_wkb(car);
9
10     ps.subscribe(IN_TOPIC)?;
11
12     loop {
13         let message = ps.receive()?;
14         let track: Tracks = prost::Message::decode(message.as_slice())
               .unwrap();
15         if is_pedestrian(&track) {
16             println!("skipping, pedestrian!");
17         } else {
18             let matched_points = matcher::match_track_on_centerline(&
                   track, &bike_centerline, &car_centerline);
19             let matched_track = track_builder::build_matched_track(&
                   track, matched_points, new_trace_in_entry);
20             let mut buffer = Vec::new();
21              prost::Message::encode(&matched_track, &mut buffer).expect
                    ("Err: failed to encode message");
22              pubs.publish(buffer.as_slice(), OUT_TOPIC);
23          }
24      }
25      Ok(())
26  }
```

Listing 12: Main flow of the map matcher of the Slight implementation

Configuration The services in the Slight implementation are configured within the `slightfile.toml` 13. As the code itself does only contain the interfaces, the actual implementation is also set within this configuration file. Every implementation for an interface is configured within a `[[capability]]` section like in line 3 and 13. The resource in line 4 defines is the identifier for the used implementation. As the documentation of Slight does not include a list of those identifiers, therefore those need to be looked up within the source code[46] of the project. They can be fond in the `lib.rs` files of the different interfaces, and always start with the name of the interface itself. As an example, the interfaces for the key value store are `kv.filesystem, kv.azblob, kv.awsdynamodb, kv.redis`. Line 5 and 15 is defining the names of the implementation. This is the name, which is getting referenced from the application for selecting the interface and loading the configuration, as seen in line 2 to 4 from the example code **??**. In line 6 to 12 the configuration for the actual interface is set. This set of config can be different for each implementation.

```
1   specversion = "0.2"
2
3   [[capability]]
4   resource = "pubsub.confluent_apache_kafka"
5   name = "my-pubsub"
6       [capability.configs]
7       CAK_ENDPOINT = "localhost:9092"
8       CAK_SECURITY_PROTOCOL = "PLAINTEXT"
9       CAK_SASL_MECHANISMS = "PLAIN"
10      CAK_SASL_USERNAME = ""
11      CAK_SASL_PASSWORD = ""
12      CAK_GROUP_ID = "mapmatcher-group"
13  [[capability]]
14      resource = "kv.redis"
15      name = "graphs"
16      [capability.configs]
17      REDIS_ADDRESS = "redis://127.0.0.1:6379"
```

Listing 13: Example for the config of a Slight component

Scaling Scaling, in this Slight-based approach, is almost identical to the one in the reference implementation. This is because both using the same broker for the messaging. Therefore, the most important thing is, that every consumer has its partition on the broker.

46. https://github.com/deislabs/spiderlightning/tree/main/crates

Because WebAssembly doesn't support multithreading, multiple instances of every service need to be started, instead of configuring the number of threads to allow concurrent use.

3.4 Measuring the latencies

The main performance metrics for stream processing pipelines are the throughput and the latency. For the chosen task within the C-ITS domain, where data arrives in a continuous stream, the throughput is the number of messages per second. The latency is the time it takes to process a message. In the chosen use case, where there will be warnings generated from incoming messages, a high latency will lead to warnings of no use, since the accident might already have happened. This means the latency is the most important metric. The throughput defines the amount of messages the pipeline can handle before the performance of the pipeline degrades, and the latency exceeds a defined amount of time.

Latency A common way of measuring the latency is to record the time of arrival of the message and the time when the output is sent. This approach can be used for an end-to-end latency measurement. However, since stream processing is a distributed system, the latency of the single services is also a relevant metric, and can't be measured by just comparing the inbound and outbound time. The latency for the unique service can be measured with the following approaches.

- When using a single message broker like Apache Kafka, the latency of the individual services can be measured by taking the timestamps of messages to arrive at the broker and the timestamps of the messages sent to the broker.
- if there are different message brokers which might use different clocks, or use other mechanisms to add the timestamps of receiving and sending messages, or if they don't store the timestamps at all, the former approach is not viable. For these cases, the timestamps can be gathered within the services, when the messages are received and when they are sent. In this case, it is important that the clock of the services is synchronized.
- another approach is to use distributed tracing systems like Opentelemetry[47], which can be used to gather information from the different services via HTTP or gRPC requests. In this case, the different services needs to be instrumented. This can be done by using the OpenTelemetry instrumentation. This way, every service sends a request to the OpenTelemetry service, which can then collect the information and build traces. Those traces represent the latency of the services, and also the end-to-end latency of the whole pipeline. It is the preferred approach, since it uses a standardized format and is already implemented in a wide range of frameworks. Downsides of this approach are, that an additional network overhead is generated for the HTTP or gRPC calls to the trace collectors and that the code of the applications might need to be adopted for more complex use cases.

47. https://opentelemetry.io/ visited at 22.11.22

For this thesis, the distributed tracing is not viable, since the inbound messages are not unique and cannot be referenced to the outbound messages. This is because the ID of the incoming C-ITS messages only provide the ID of the device which sends the message and the objects which are inside the message cannot exceed 256 different IDs which are then reused. Another reason for not choosing the distributed tracing is, that the gRPC or HTTP calls are not available directly in Webassembly. Therefore, the second approach, where the service is measuring the inbound and outbound time for each message, has been used. This measuring behavior has been implemented directly into the message deserializer and serializer. This way, the measuring take place as early and as late as possible. Those timestamps are then added as a field into the messages themselves. This way, every message contains a history of every step in the processing pipeline with the needed timestamps.

The chosen approach allows implementing tracing without touching the business logic or introduce external services. The downside is the additional size overhead which is added to every message.

Throughput The throughput will be tested by increasing the frequency of messages until the pipeline cannot process the messages within the defined maximum latency anymore.

4 Evaluation

4.1 The test setup

For the evaluation, a set of 3,323 CPMs is used as datasets. Those messages will be sent to a MQTT local MQTT broker from which a message decoder, similar to the one on the actual data ingestion infrastructure, consumes the messages and translates them to the internal Protocol Buffer representation. Unlike the actual service, the one from the test setup can produce the resulting messages either to an Apache Kafka log, a Redis Pub/Sub channel or a Nats stream. Another additional function of this message translation service is, that it adds a first tracing timestamp to the messages directly before the messages are sent. This initial tracing timestamp will then be compared to the timestamp which gets added to the tracing object when the map matching service sent the final message to the broker.

Data representation The test data representation of the recorded test dataset is stored as base64 encoded binary file. In this binary representation, the first four bytes are defining the length of the next CPM, the following 8 bytes representing the timestamp of the message. From the remaining bytes, the length which has been represented by the first bytes are representing the actual message. The message is followed by the length for the next message. This representation is small and easily workable, moreover, it is language-agnostic. The timestamps inside the binary file are used for the test service to send the messages with the original frequency or a modified version of the frequency.

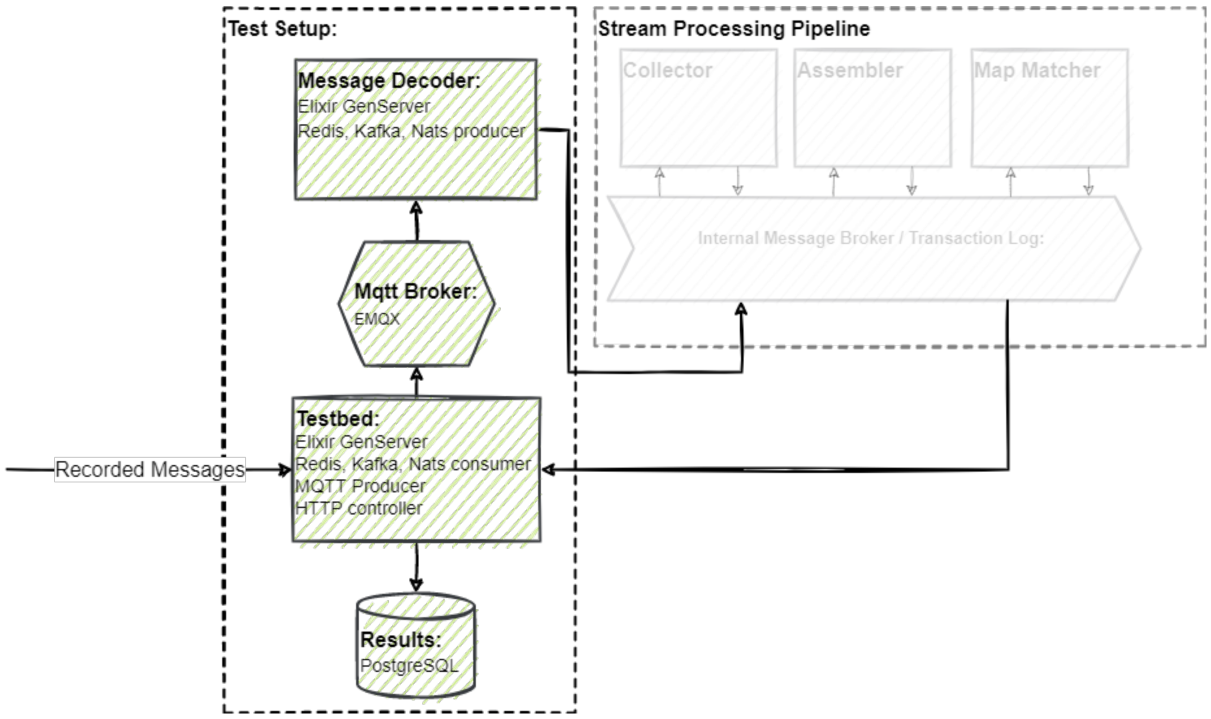


Figure 13: The test setup

Testbed The testbed consists of four parts. One that can receive recorded messages, one which is connected to a MQTT broker, one that consumes from the individual message broker of the different pipelines, and one which is connected to a PostgreSQL[48] database. The first component received the base64 encoded binary messages and starts the test scenario for the recording. The second part prepares the messages which are parsed from the recorded binary file and sends them either with the original delta times or with an altered frequency to the MQTT broker. The third part consumes the messages from the topic, channel, or stream in which the map matching service write its results. The final service takes the consumed messages, extracts the tracing information and persists them into the connected PostgreSQL database. The persisted data can then be used for the evaluation of the results.

[Message decoder] The message decoder subscribes to the MQTT topic, which receives the messages from the testbed. It then decodes the CPM, which is contained within the message wrapper, and translates it to the internal Protocol Buffer representation of the CPM. Then the Protocol Buffer-based CPM is wrapped in another message wrapper which contains the initial tracing information to the message. The generated message is then sent to the configured messaging system, from where the pipeline will consume the message.

Test scenarios The following Test scenarios will give an answer, to the question, if server-side WebAssembly is suitable for the given stream data processing task.

48. https://www.postgresql.org/ visited 11.11.22

- For the evaluation, the recorded messages are first replayed with the original frequency. This results in 16 messages per second, eight per camera, for the collector, up to theoretically 2048 messages per second for the assembler and the map matcher. This maximum value will not happen, since this would mean, that every CPM contains 128 detected objects. A realistic assumption for the number of the detected objects within a message is much lower. An assumption is an average of eight detected objects per message. This would lead to 128 messages per second for the assembler and the map matching service and result in overall 272 messages per second. Although, since the data is an actual recording, the number of detected objects will not be spread evenly and the number of messages might be lower, or even at zero for a short time span.
- A second evaluation takes the same recording of messages and plays them back twice as fast. This should lead to about 544 messages per second for the whole pipeline.
- A third run will again double the frequency for an approximately, 1088 messages per second. The number should be high enough that the pipeline will be at or above its limit when it is only deployed on one server.
- In the fourth scenario, the messages in the original pace are processed on a scaled down system.
- In addition to the four scenarios, there is a brief comparison of the package size, of the modules.

The first test case displays the capability of the pipeline for the real use case. For the research project, in which the pipeline is used, an overall latency lower than 500 milliseconds is acceptable. If the 95 percentile of the latencies is higher than this threshold, the pipeline is considered as not viable. In the second and third test case should show if, and by how much, the latency will increase under higher load. The tests are repeated three times to rule out possible external interference. The fourth scenario, as well as the comparison of the sizes from the packages, shows the hardware requirements for the different approaches.

Hardware The test platform is on a desktop, equipped with an AMD Ryzen 3600 six core CPU and 32 GB of DDR4 ram. The testbed is a lightweight Elixir[49] application which will also run on the same instance as the actual pipeline. For the final scenario, the amount of ram as well as the number of available CPUs is reduced.

4.2 Latencies with the original data

In the first evaluation, messages are processed in the original frequency. A test run takes about four minutes and gets repeated three times. This test case should display, if a selected approach is suitable for the actual workload. The latency of a pipeline is defined to be acceptable if the

49. https://elixir-lang.org visited 11.11.22

95th percentile is below 500 milliseconds. The three runs are always plotted into the same figures. The first run in green, the second in blue and the third in orange. In a second part of this evaluation, the latencies are inspected for the individual components.
All latencies are measured in milliseconds, and microseconds are rounded to the nearest full millisecond. This means 1.5 milliseconds will be shown as 2 milliseconds in the table. Values lower than 0.5 are shown as 0 milliseconds.

4.2.1 Entire platform

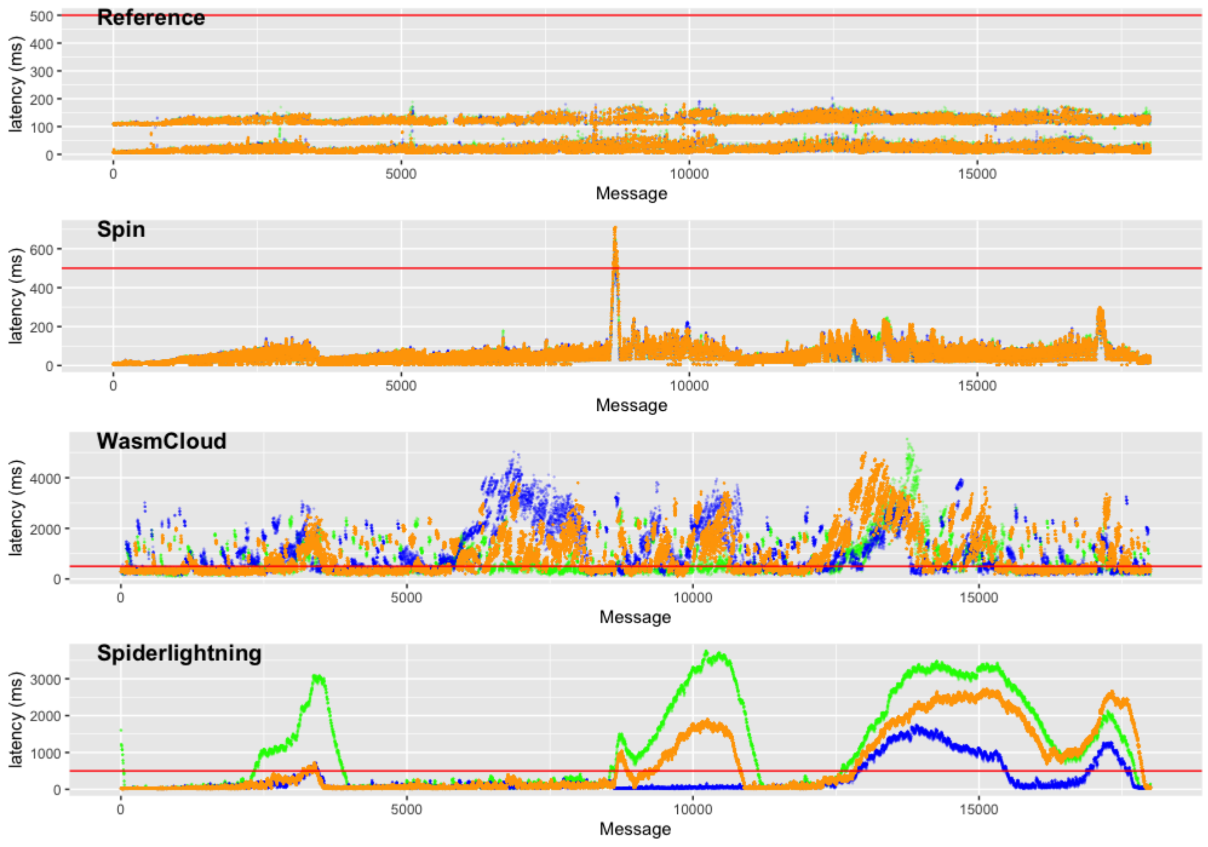


Figure 14: Evaluation of the entire pipeline, original frequency

Reference The Spring Cloud Streams implementation, which acts as reference, is the first evaluated pipeline. The pipeline is configured to use a single thread for the collector, the assembler, and the map matcher are running multithreaded with four threads each. As figure 14 shows, the reference implementation can meet the requirement by processing every message significant faster than required. The latency for the 95th percentile is 154 milliseconds, as stated in table 1. The slowest processed message took 204 milliseconds, which is still two times faster as the requirement. Furthermore, there is a pattern visible in figure 14, where there are always faster processed messages, followed by slower ones.

	Reference	Spin	WasmCloud	Spiderlightning
min latency	4 ms	2 ms	76 ms	10 ms
max latency	204 ms	711 ms	5533 ms	41 ms
mean latency	64 m	66 ms	954 ms	19 ms
standard deviation	53 ms	52 ms	850 ms	2 ms
50th percentile	32 ms	59 ms	550 ms	19 ms
95th percentile	139 ms	139 ms	2785 ms	23 ms
99th percentile	154 ms	236 ms	3758 ms	25 ms

Table 1: Results for the entire pipeline, using the original frequency for the messages

The reason for this is not clearly visible in the figure, but the majority of the messages can be completely processed within roughly 35 ms. This pattern leads to a standard deviation of 53 milliseconds.

Spin The Spin implementation is the second evaluated platform. The pipeline uses a KeyDB instance which utilized four threads, instead of a single threaded Redis service. This should reduce a possible bottleneck at the key value store when running several WebAssembly modules concurrently. As visible in the tab;e 1, half of the messages are handled within 60 milliseconds, this value is almost two times higher than the one from the reference implementation. This table also shows that the 95th percentile is identical to the reference implementation, with a value of 139 milliseconds. The same applies to the mean latency, which is in the range of 60 milliseconds. Although those measures are almost identical, the figure 14 highlights, that the latencies of the WebAssembly-based approach showing a different pattern than the Spring Cloud Stream approach. The messages are getting processed slower as the number of processed ones grows. Then, once a peak is reached, the messages are getting processed faster and the pattern repeats itself. Across all three runs of this pipeline, there were 152 messages which were processed slower than 500 milliseconds, which results in 0.28% of messages which are not meeting the requirements. Since this is below the defined threshold, the pipeline can meet the requirement from this scenario.

WasmCloud The WasmCloud implementation is the third evaluated platform. As the figure 14 clearly shows, this approach is not suitable for the task. In the three test runs, 28809 messages are processed slower than the required 500 milliseconds. These are 53.35% of the messages. About a third of the requests are even slower than one second. The fastest processed messages took 76 milliseconds, which is already slower than the slowest messages from the Slight implementation. The mean latency is with almost one second between 14 and 49 times slower than the other implementations. Because of this unsatisfying results, the WasmCloud implementation is not evaluated further.

Spiderlightning/Slight The Slight implementation is the last evaluated approach. The approach is using the same configuration as the reference implementation, which means, the collector runs on a single instance while the assembler and the map matcher are utilizing four instances each. The figure 14 shows that this approach is significant faster than the other WebAssembly implementations. The table 1 shows that the Spiderlightning implementation is well suited for this given test scenario. Except for the minimum latency metric, it is even faster than the reference implementation. With a 95th percentile latency of 23 milliseconds, and a standard deviation of roughly 2 milliseconds, it also is more predictable than the other approaches. The maximum latency is with 41 milliseconds, almost 5 times lower than the reference implementation and 37 times lower than the one from the serverless WebAssembly approach. The minimum latency is with 10 milliseconds, relative slow. It is 5 times slower than from the Spin implementation, and 2.5 times slower than the reference.

The result of the first evaluation shows, that two of the three observed WebAssembly implementations a viable for the given use case. WasmCloud could not meet the requirement. The approach, which used Spin, showed similar latencies to the reference implementation, except for a few slow messages, the Spiderlightning approach showed a much better result, where all messages with an almost constant latency for every message.

4.2.2 Individual components

For the inspection of the individual components, only the first run of the test is used. This is because the previous figures show that the different runs are showing almost the same behavior.

	Collector			Broker			Assembler			Broker			MapMatcher		
	R	F	S	R	F	S	R	F	S	R	F	S	R	F	S
min latency	0	0	0	0	0	2	0	0	0	0	0	5	0	0	0
max latency	10	6	4	44	20	21	115	5	2	84	633	19	83	38	6
mean latency	2	1	0	2	5	7	43	1	0	2	49	7	12	6	0
50th percentile	2	1	0	2	2	7	2	1	0	1	42	7	9	94	0
95th percentile	3	2	1	6	6	8	102	2	1	7	114	9	30	11	1
99th percentile	4	3	1	9	9	9	104	3	1	32	213	10	41	14	2

Table 2: Results for the individual components, using the original frequency for the messages. Values in milliseconds. R = Reference, F = Spin, S = Slight

Reference From the components of the reference pipeline, see the first column in figure 15 and in the table 2, it can be observed, that most of the latency comes from the assembler as well as the map matcher. Furthermore, the origin of the pattern where fast messages are followed by slow ones can be found within the assembler service, where at least half of the messages are processed within 2 milliseconds, and the other half somewhere in the range of 90 to 115 milliseconds. It can also be observed that the communication from

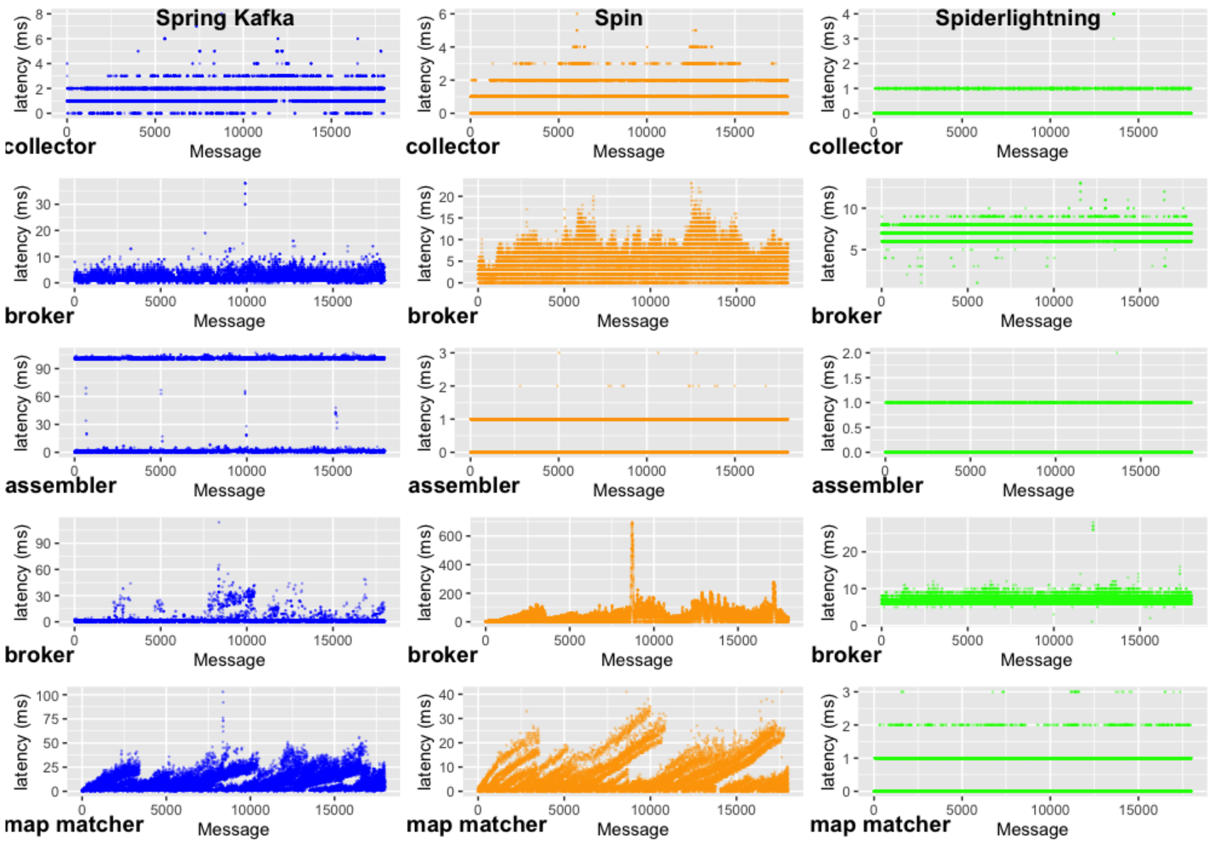


Figure 15: Individual components, original frequency

the assembler to the map matcher has slower transmissions more often. As expected, the collector is the fastest of the three services and can handle every message within 10 milliseconds. The assembler, however, was expected to be faster than the map matcher, which was not the case. The transmission latency between the services has some noticeable spikes, which leads to some slower processed messages. Furthermore, It is visible that, except for the assembler service, the messaging causing a big part of the overall latency.

Spin When looking at the table 2 at the different WebAssembly modules from the Spin implementation, it can be observed, that the gross of the latency comes from the message broker. While the services itself are faster than the ones from the reference implementation, the latencies between the services are much higher. Figure 15 shows that the pattern of the latencies between the assembler and the map matcher closely resemble the one from the entire pipeline 14. It should be noted that the time between services includes the publishing of messages as well as the receiving of messages and the instantiation of WebAssembly services, since the timestamps are added only directly after receiving and before sending the message. While the speed of the collector is comparable with the reference implementation, the assembler is almost 50 times faster and the map matching service is about two to three times faster. Unlike the reference implementation, the map matcher is the slowest of the three components, which is the expected behavior of the service.

Spiderlightning/Slight When looking at the third column figure 15 for the Slight implementation, it is noticeable, that the three services are even faster than the Spin ones. The 99th percentile for the collector, as well as the assembler, are three times lower than their Spin equivalent. For the map matcher, the latency is 14 times lower. This difference is most likely because the Spiderlightning implementation is stateful and does not need to read the street geometry on every message. The main driver of the latency is, like for the other WebAssembly-based approaches, the messaging between the services. In the table 2 can be seen that the mean time spent on messaging is about 18 times more than the actual processing time of the messages. It is also interesting, that the messaging is faster than the Spring Cloud Streams implementation, even if they are both using the same messaging system. This implies that the higher latencies observed between the assembler and the map matcher in the reference implementation are due to the map matcher being unable to consume messages faster. The similar pattern in the charts for the map matcher and the second broker might also be an indicator for this assumption.

The Wasmcloud implementation is no longer evaluated, since it does not meet the requirements of the use case.

When looking at the individual components, it is noticeable, that the messaging between the Spin components, which uses Redis Pub/Sub, is significant slower than for the approaches that are using Apache Kafka. Furthermore, it is noticeable that the actual processing time is faster in the Spin and Spiderlighning implementations. The evaluation also leads to the assumption, that the processing time of the map matching service from the reference implementation, cannot process the incoming messages as fast as Apache Kafka would allow to.

4.3 Latencies with a two times higher frequency

In the second evaluation, the same data is received at a two times higher frequency. This test should show if the required latency can still be reached if the number of messages per second is doubled. While the first evaluation shows if the used approach is good enough for the actual use case of the research project, this evaluation should show, if the implementation would be viable if more than a single intersection is getting processed at the same time. This evaluation is again split into a view at the entire platform and a view at the individual components. Latencies are again rounded to the nearest full millisecond.

4.3.1 Entire platform

Reference With the doubled frequency, the performance of the reference implementation improved compared to the first use case. As in figure 16 can be seen, the pattern, where fast requests are followed by slow ones over the whole test run, does not appear to happen

	Reference	Spin	Spiderlightning
min latency	5 ms	2 ms	10 ms
max latency	248 ms	56750 ms	62 ms
mean latency	27 m	16068 ms	22 ms
standard deviation	18 ms	14347 ms	4 ms
50th percentile	23 ms	13768 ms	22 ms
95th percentile	54 ms	44654 ms	29 ms
99th percentile	109 ms	56098 ms	33 ms

Table 3: Results for the entire pipeline, using the doubled frequency

anymore. As table 3 shows, the average latency for the processing is two times faster than with the original frequency. The reason for this behavior is not clear. With that behavior not appearing anymore, the 95th percentile is more than two times lower than in the previous test case 1. The value for the median latency is comparable with the earlier test. This can be observed in the figure 14, as about half of the messages were processed as fast as in this test. This also reduced the standard deviation from 53 to 18 milliseconds. The only measurements, which are worse than with the original frequency, are the minimum latency and the maximum latency.

Spin Unlike the reference implementation, Spin performs worse when the frequency for the messages is doubled. This is the expected behavior. Figure 16 shows that the latency is increased with every new message and the required maximum latency is no longer met. The pattern lead to the assumption, that the pipeline cannot process messages as fast as needed, which leads to a backlog of messages. With every new message, this backlog gets bigger and the time until the message gets processed increases. Table **??** shows that the average latency is already roughly 16 second, which is much higher than required. The 95th percentile, which is required to be less than half a second, is in this scenario almost 90 times higher than it should be. This behavior is investigated further at the component view.

Spiderlightning/Slight The Slight implementation performs with the doubled frequency, similar to the original frequency. As visible, in the table 3, the latency of the pipeline increased for every measure, except the minimum latency, where it is still 10 milliseconds. This is, like in the first scenario, slower than its peers. The average latency is about two milliseconds slower than with the original data. The 99th percentile is 8 milliseconds higher, which is again the lowest value of the compared pipelines. The required latency can be met, and not a single message is processed slower than 500 milliseconds, which can also be seen in figure 16. The standard deviation, has been raised from 2 to 4 milliseconds, which is still low, when it is compared with the reference pipeline, which has a standard variation of 18 milliseconds.

The evaluation, with an increased number of messages per second, showed that the reference

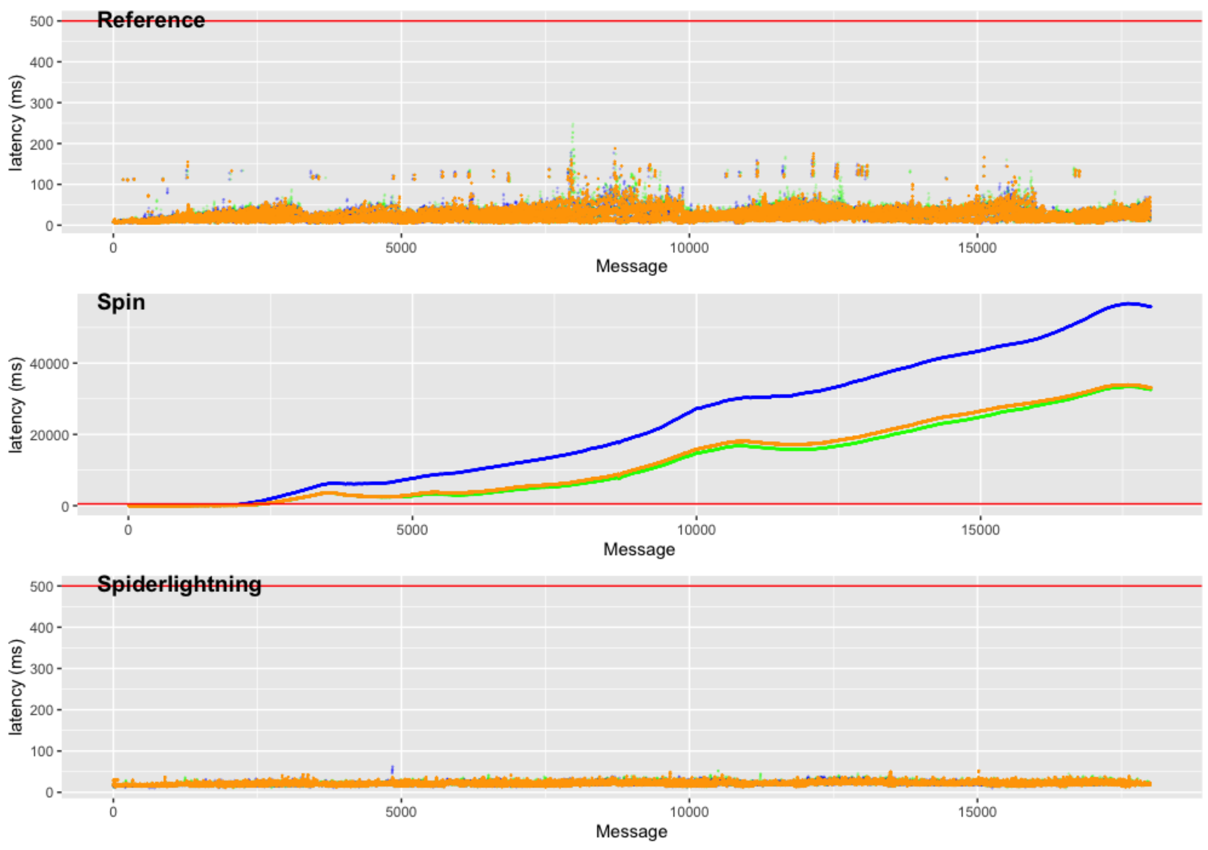


Figure 16: Evaluation of the entire pipeline, doubled frequency

pipeline is well suited for its task. It even performed better than with fewer messages per second. It has also been shown that the approach using Spiderlightning, which shared the same broker and architecture than the reference implementation, can also easily handle the task. The serverless approach, on the other hand, could not process the messages in the required time. The shown performance was significant worse than in the first evaluation.

4.3.2 Individual components

Since the results of the different runs of the test showed similar results, there is again only the first run used for the analysis of the individual components.

Reference When looking at the individual components from the Spring Cloud Stream implementation 17, it is visible that this time, the map matcher is the slowest involved service. It is the only service of the implementation which got slower with the increased number of messages. The table 4 shows that the assembler is in average about 20 times faster than with the original frequency 1. The 99th percentile, has been reduced by 96 milliseconds, while the 50th percentile is reduced only by one millisecond, compared with the lower messages per second 1. This highlights the pattern from the initial scenario, where about

	Collector			Broker			Assembler			Broker			MapMatcher		
	R	F	S	R	F	S	R	F	S	R	F	S	R	F	S
min latency	0	0	0	0	0	1	0	0	0	0	0	4	0	0	0
max latency	9	10	1	30	27	26	104	8	11	195	33404	33	166	46	16
mean latency	2	1	0	1	6	7	2	1	0	3	12209	8	15	7	2
50th percentile	2	1	0	1	5	7	1	1	0	1	9466	8	12	4	1
95th percentile	2	3	1	3	13	8	3	1	1	17	32378	14	35	21	5
99th percentile	3	3	1	6	17	9	8	1	1	39	33315	18	47	28	7

Table 4: Results for the individual components, using the doubled frequency for the messages. Values in milliseconds. R = Reference, F = Spin, S = Slight

half of the messages were unusual slow. Although, this pattern can still be seen in the plot 17 for a couple of messages. The number of those is relative low, and does not exceed one percent, as the 99th percentile is not affected by them. The messaging between the components, got slower between the assembler and the map matcher, but faster between the collector and the assembler.

Spin The view on the individual components of the Spin implementation shows, that the main part of the latency comes from the messaging between the assembler and the map matcher, figure 17. Even if the 95th as well as the 99th percentile for the map matching service is now almost two times as high as before, the services are processing the data faster than the counterpart from the reference implementation. The fast services, with the relative high latency between them, is the same behavior as with the original frequency of messages. The big difference is, that the time between publishing the message in the assembler and the subscribing in the map matcher grows for every message, up to over 30 seconds. This behavior makes this approach not usable for this use case, even if the individual services would perform well, as shown in table 4.

Spiderlightning/Slight Figure 17 shows, that at component level, the messaging between the assembler and the map matcher is the slowest part of the pipeline. Those latencies increased by 5 milliseconds from the collector to the assembler and 14 milliseconds from the assembler to the map matcher. When looking only at the 99th percentile, the time spent on messaging is almost three times higher than the processing time. Compared with the above table 2, the difference between time spent on messaging and processing is not as huge before. Table 4 displays, that the collector, as well as the assembler, are taking almost the same amount of time, as with the original message frequency. The map matcher, on the other hand, took in average more than two times as long with the doubled frequency. Although, with 2 milliseconds in average it still performed better, than its peers. Inspecting the components showed, that the bottleneck of the serverless WebAssembly approach are not the services themselves, but the part which is done between the execution of them. It is not clear, if its origins in the broker itself, or the Spin runtime. For the reference implementation as well as the Slight implementation, the component view didn't lead to new insights, compared with the observation with the original

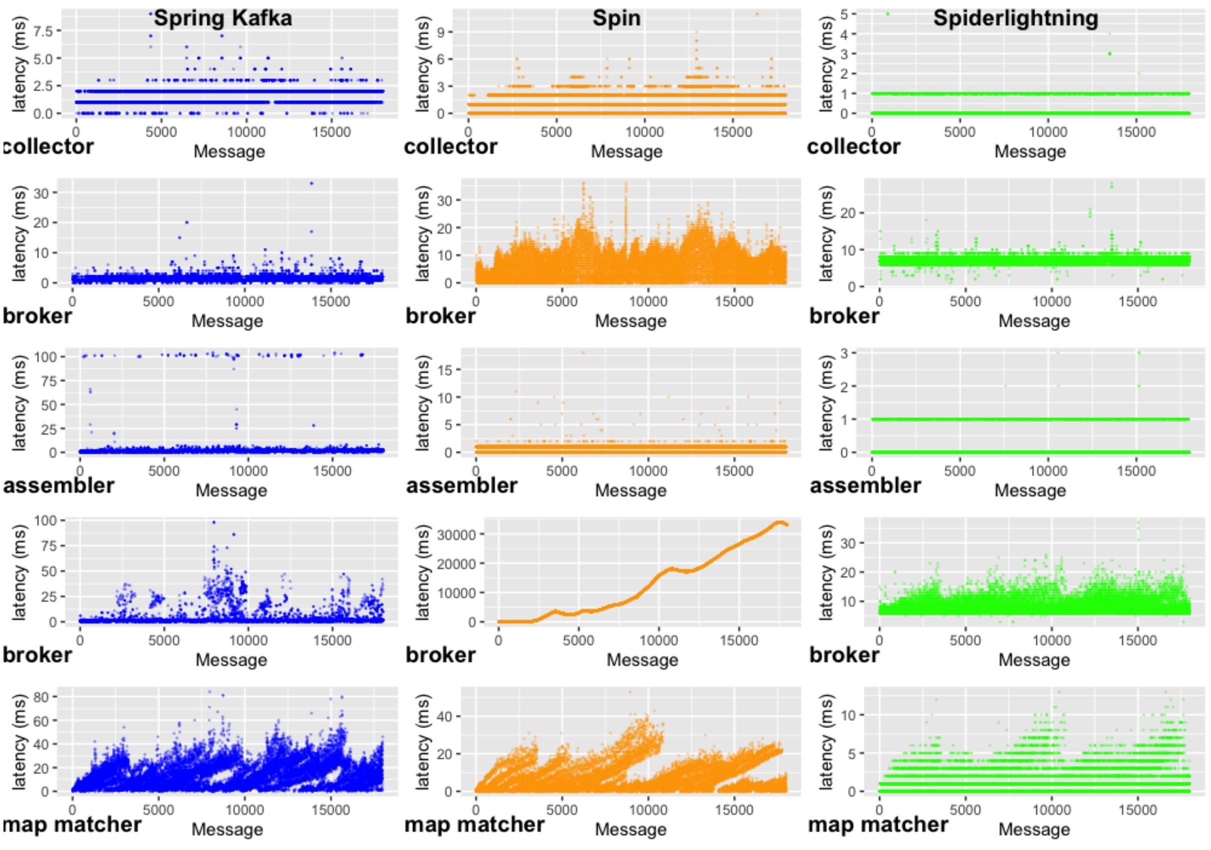


Figure 17: Individual components, doubled frequency

number of messages per second. Although it is apparent that the map matcher, which was intended to be a slow service, is actually getting slower, with an increased message frequency. This has been shown for all three remaining implementations.

4.4 Latencies with a four times higher frequency

In the third evaluation, the latencies are doubled again. This test should add more load to the pipeline. It is like before, split into a component view and a view over the entire stream processing pipeline, and the latencies are rounded to whole milliseconds.

4.4.1 Entire platform

Reference With the quadrupled frequency, the reference platform is unable to meet the requirement that the 95th percentile is below 500 milliseconds, as shown in table 5. The average latency is with 411 milliseconds, already close to the targeted maximum value. The 50th percentile shows, that at least half of the messages are processed reasonable fast, with only 36 milliseconds. The standard deviation is with 840 milliseconds, already greater than the allowed latency. When looking at figure 18, it can be seen that many messages

	Reference	Spin	Spiderlightning
min latency	5 ms	3 ms	11 ms
max latency	4680 ms	96688 ms	1571 ms
mean latency	411 m	36583 ms	29 ms
standard deviation	840 ms	26621 ms	49 ms
50th percentile	36 ms	35664 ms	25 ms
95th percentile	2590 ms	81782 ms	37 ms
99th percentile	3235 ms	95693 ms	48 ms

Table 5: Results for the entire pipeline, using the quadrupled frequency

are getting processed slower by every iteration. This is most likely because of the growing message size for single trajectories, and should also be visible at the component view within the map matcher.

Spin As expected because of the earlier results for the serverless Spin approach, the latency increased again with more messages per second. As figure 18 displays, the pattern looks like in the evaluation before, but the latency grows faster. In the table 5 can be seen that the latency almost reaches 100 seconds.

Spiderlightning/Slight The Spiderlightning approach could still meet the requirements, even if the messages per second are four times as much as recorded. As 18 shows, there is a spike at the beginning of one of the three test runs, but the number of messages in these anomalies do not affect the 95th percentile, or even the 99th percentile, of the pipeline. The 99th percentile latency is for this scenario below 50 milliseconds, and about 67 times faster than the reference implementation. It is even two times faster when it is compared with the reference from the previous scenario 3. Compared with the same implementation from the second scenario, it is about 50% slower. 5.

Looking at the results, of the evaluation with four time as many messages per second as in the original scenario, shows that the WebAssembly-based streaming pipeline which uses Apache Kafka is the only one of the evaluated approaches which can handle this number of messages, on a single server. The Spin implementation got much slower again, which was expected, given the earlier results. The reference platform, also, couldn't handle to process the messages in time. Although it performed significant better than the serverless approach.

4.4.2 Individual components

Reference In the reference implementation, when looking at table 6, it can be observed, that the reason for the high latency with the quadrupled message frequency is caused by the map matcher and is also showing within the messaging between the assembler and the map matcher. In figure 19, it can be seen, that the high latencies within the messaging are

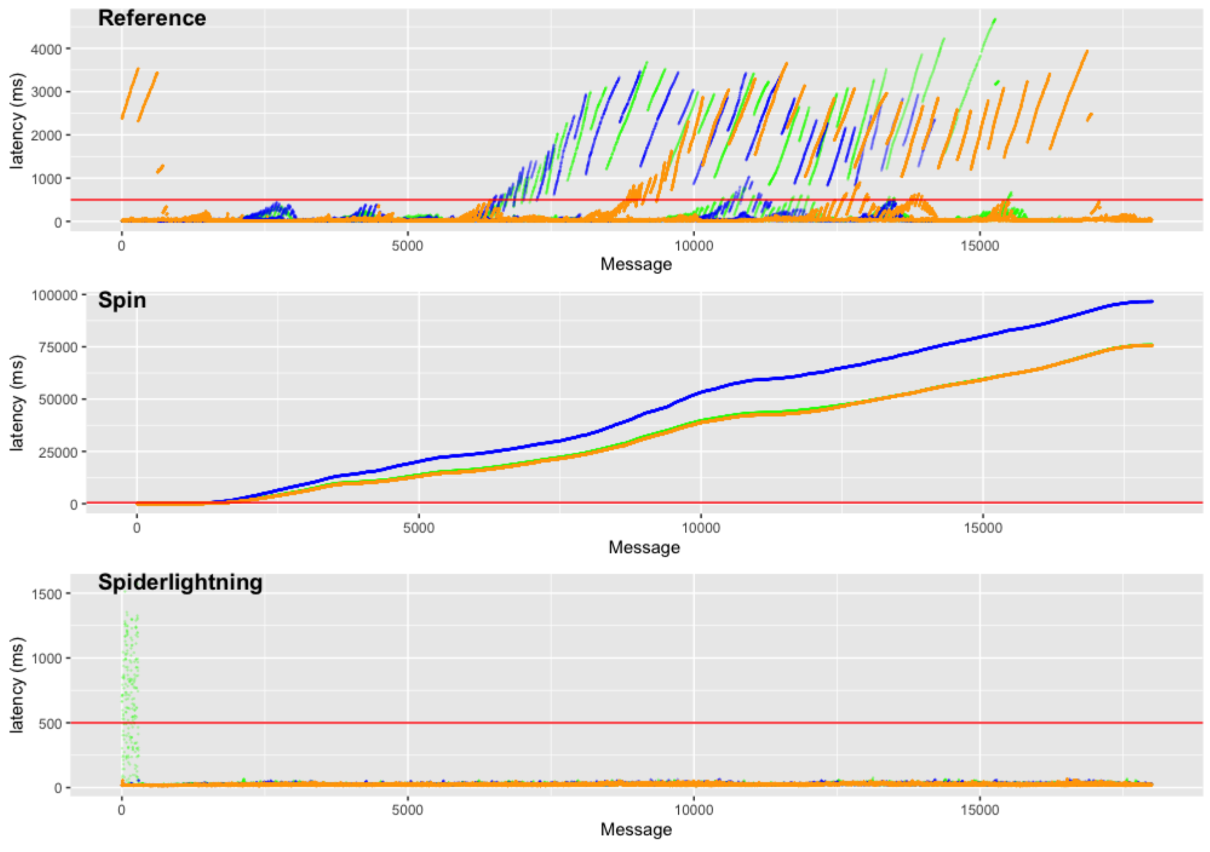


Figure 18: Evaluation of the entire pipeline, quadrupled frequency

	Collector			Broker			Assembler			Broker			MapMatcher		
	R	F	S	R	F	S	R	F	S	R	F	S	R	F	S
min latency	0	0	0	0	0	1	0	0	0	0	0	2	0	0	0
max latency	10	8	3	31	33	25	103	9	12	3155	75885	45	3137	43	24
mean latency	2	1	0	2	5	8	2	1	0	243	329223	10	174	7	2
50th percentile	2	1	0	2	4	8	1	1	0	3	32063	8	18	4	1
95th percentile	4	3	1	6	11	9	3	1	1	1533	72839	18	1185	20	7
99th percentile	6	3	1	13	14	13	7	1	1	2515	75788	25	2090	26	10

Table 6: Results for the individual components, using the quadrupled frequency for the messages. Values in milliseconds. R = Reference, F = Spin, S = Slight

due to the messages not being processed fast enough from the map matching service. The map matcher processed messages in average within 174 milliseconds, which is already a sign, that the queue for this service cannot be processed in a reasonable time.
Since the pipeline is generating about 20000 messages as output all together, and the messages with the quadrupled frequency are all send within a timespan of about 60 seconds, this means that the map matcher need to be able to process roughly 333 messages per second. If it runs with four concurrent jobs, this leads to about 84 messages per second

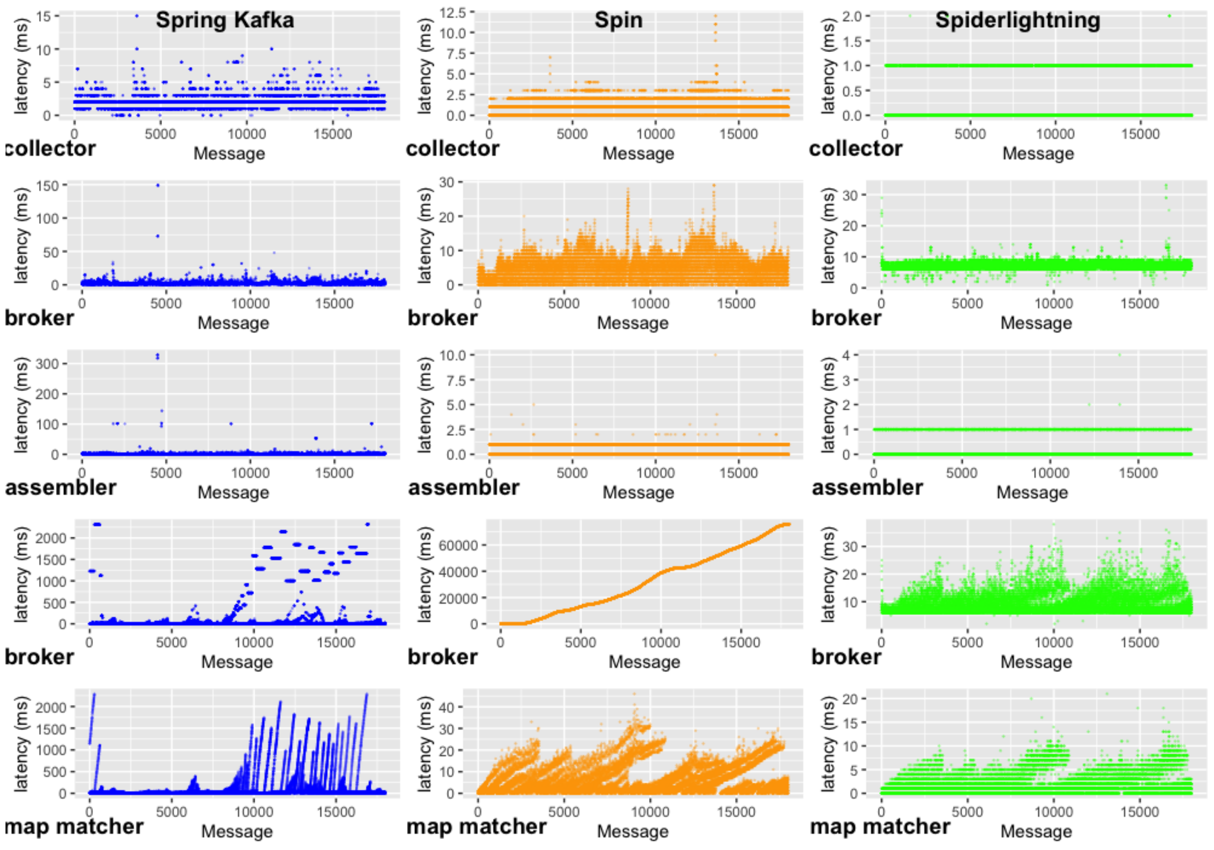


Figure 19: Individual components, quadrupled frequency

per job. Therefore, the processing time per message needs to be lower than about 12 milliseconds to keep up with the ongoing flow of incoming messages.
Therefore, the queue within the broker is growing with every new message and as a result, the time before messages are pulled from the broker is increased. This is also visible within the table 6, as well as in the chart 19.

Spin The component view for the Spin implementation shows, again, that almost the entire latency comes from the messaging between the assembler and the map matcher. It can also be observed in figure 19, that this latency is growing with every new message. The other services performed almost identical to the test run, with only half as many messages per second as visible in table 6.

Spiderlightning/Slight In the Spiderlightning implementation, the results 6 are looking similar to previous observations with this approach. The collector, as well as the assembler, performed almost identical as before, while the latency for the map matcher and the messaging time has been increased by about 43%. The ratio between actual processing time and time spent on the messaging between the components is about the same as in the second evaluation, where the messaging takes about three times as much time as the processing.

At component view, the scenario with the quadrupled message frequency highlighted again, that the bottleneck for the Spin approach is not caused by the processing speed within the individual services. It is likely, that this behavior comes directly from the Spin framework itself, or the Redis pub/sub broker. The origin of the too high latency for the reference, could have been identified. It is the map matching service, where messages are not processed fast enough. For the Slight implementation the component view displayed, that even with the higher frequency for the messages, the broker is the slowest part of the pipeline.

4.5 Scaled down system

In the last evaluation, the stream processing pipelines are running at multiple scaled down setups. For this scenario, the system resources have been limited to a single CPU core and a two CPU cores with 4 GB of Ram each. This should emulate the usage of an edge computing device or a low-cost cloud instances. The 4 GB of Ram have been chosen, since the services of the reference implementation wouldn't start with less memory. The single core has been chosen as it is the minimum possible size and the dual-core setup because this is the setup of the Amazon AWS large EC2 instances[50]. The test runs have been executed again three times per pipeline.

4.5.1 Single core

For the evaluation of the single core performance all three runs are visualized, but only the numbers from the third run (orange points) are taken for the table 7, as it has the least outliers. When looking at the figure 20, it can be observed, that not a single implementation could meet the requirements. The reference implementation is the slowest, followed by the serverless Spin implementation. The Spiderlightning-based stream processing pipeline performed a little better, but there is the same pattern of slow messages visible as in the Spin implementation.

	Spring Cloud Streams Kafka	Spin	Spiderlightning
min latency	7 ms	2 ms	13 ms
max latency	36303 ms	4674 ms	2720 ms
mean latency	7522 ms	1450 ms	722 ms
standard deviation	10628 ms	1620 ms	863 ms
50th percentile	1964 ms	411 ms	200 ms
95th percentile	32614 ms	4270 ms	2480 ms
99th percentile	35254 ms	4527 ms	2613 ms

Table 7: Results for the entire pipeline, running on a single CPU core

When looking at the individual components of the pipeline in figure 21, it can be observed, that the bottlenecks are the same as before for the serverless implementation and the Spring Cloud

50. https://aws.amazon.com/ec2/instance-types visited 11.20.22

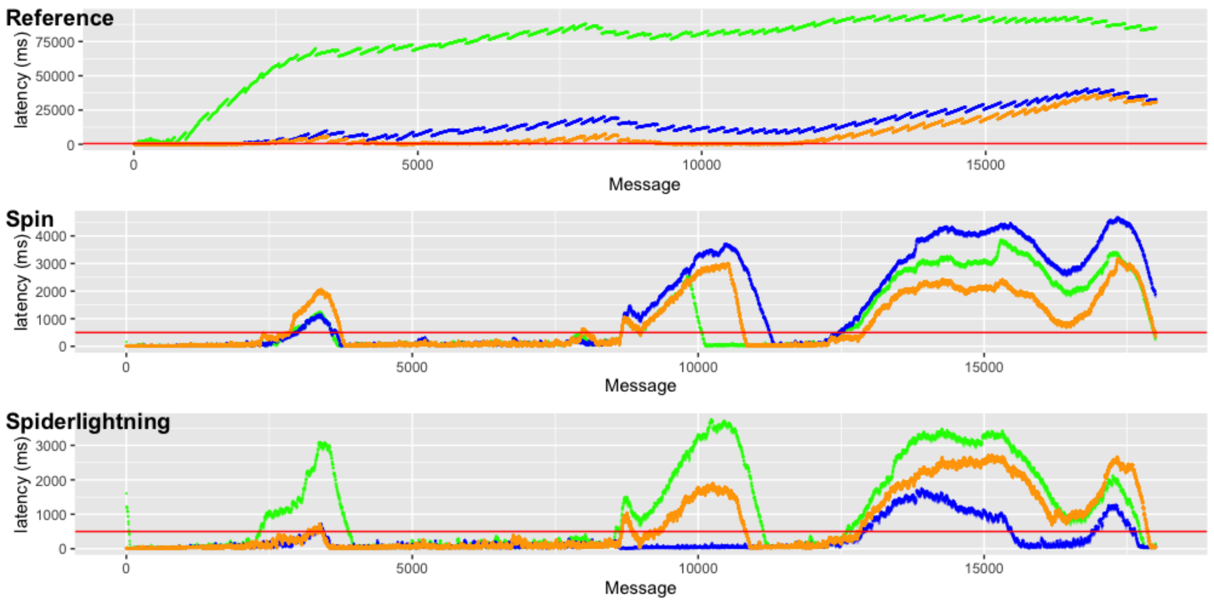


Figure 20: Evaluation with only a single core CPU, original frequency

Streams one. In the latter, the map matcher service causes the slow processed messages. For the Spin implementation, the messaging between the assembler and the map matcher slows down the whole pipeline. An interesting find, is that with such a limited number of resources, the map matcher of the Spiderlightning approach, is noticeably slower than the one from the Spin pipeline.

4.5.2 Dual core

Increasing only the number of CPU cores, from one to two, reduces the latency significantly. For both implementations, Spiderlightning and the reference implementation, the 95th percentile is lower than 500 milliseconds as seen in table 8. The requirements were not met by the Spin implementation, despite the 95th percentile being only a fifth of the previous percentile. For the reference implementation, the 95th percentile is less than a 30th of before. Spiderlightning is approximately 48 times faster when using a second CPU core, compared to the only use of a single CPU core.

At component level, there are no surprises, as it can be seen in figure 23. For the reference pipeline, the map matcher is the slowest component and for the other WebAssembly based ones the messaging causes the majority of the latency.

Since increasing the count of the CPU cores leads to a great reduction of the latency, the reference as well as the Spiderlightning pipelines are tested with the doubled and quadrupled message frequency as well.

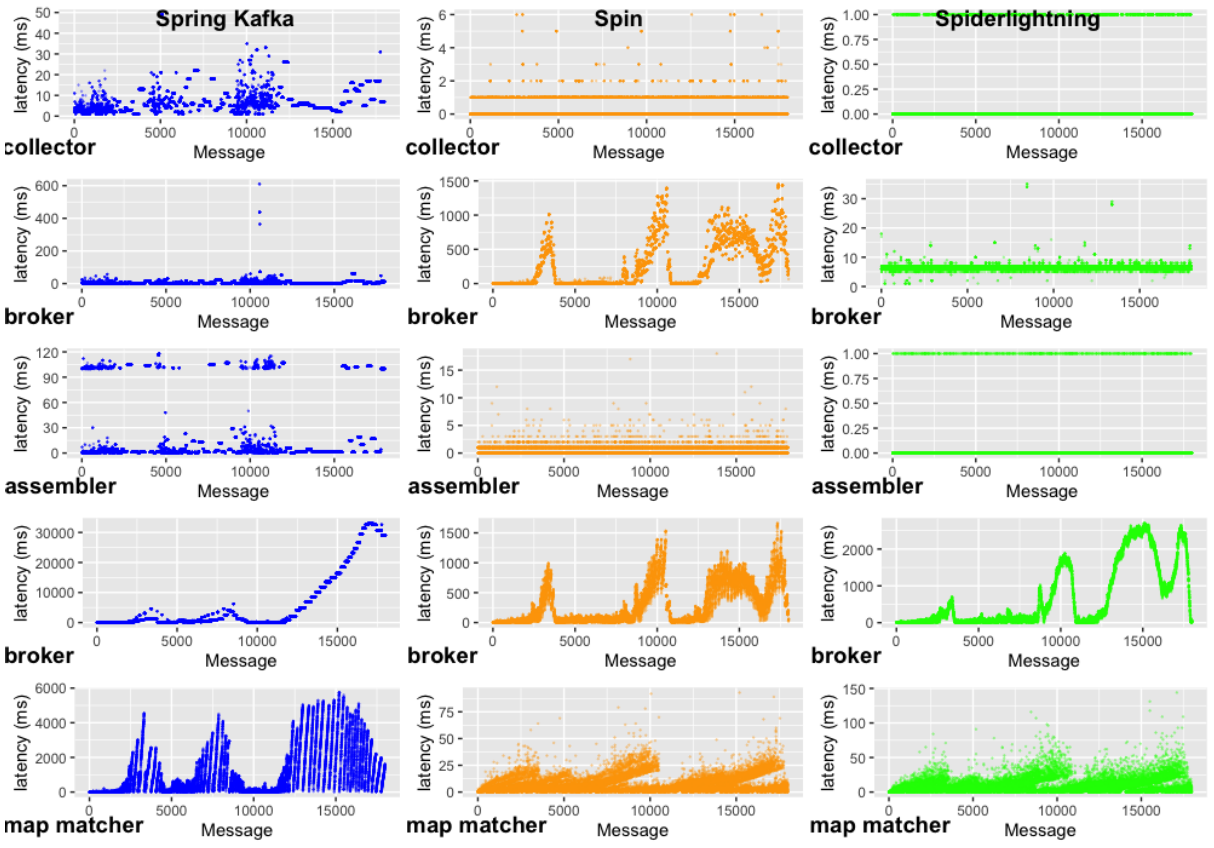


Figure 21: Single core test run, original frequency

	Spring Cloud Streams Kafka	Spin	Spiderlightning
min latency	4 ms	2 ms	10 ms
max latency	604 ms	1267 ms	107 ms
mean latency	97 ms	175 ms	29 ms
standard deviation	76 ms	260 ms	10 ms
50th percentile	78 ms	82 ms	27 ms
95th percentile	220 ms	916 ms	49 ms
99th percentile	370 ms	1182 ms	60 ms

Table 8: Results for the entire pipeline, running on two CPU cores

4.5.3 Dual-core with higher frequencies

When increasing the message frequency by the factor of two, and four, the Spiderlightning approach could still meet the requirement. As visible in figure 24, the reference implementation couldn't finish the run, as the latency increased until the system crashed after about 2,000 messages. Except for a small spike at the beginning, both test runs that utilized Spiderlightning, managed to perform well, even with the scaled down system.

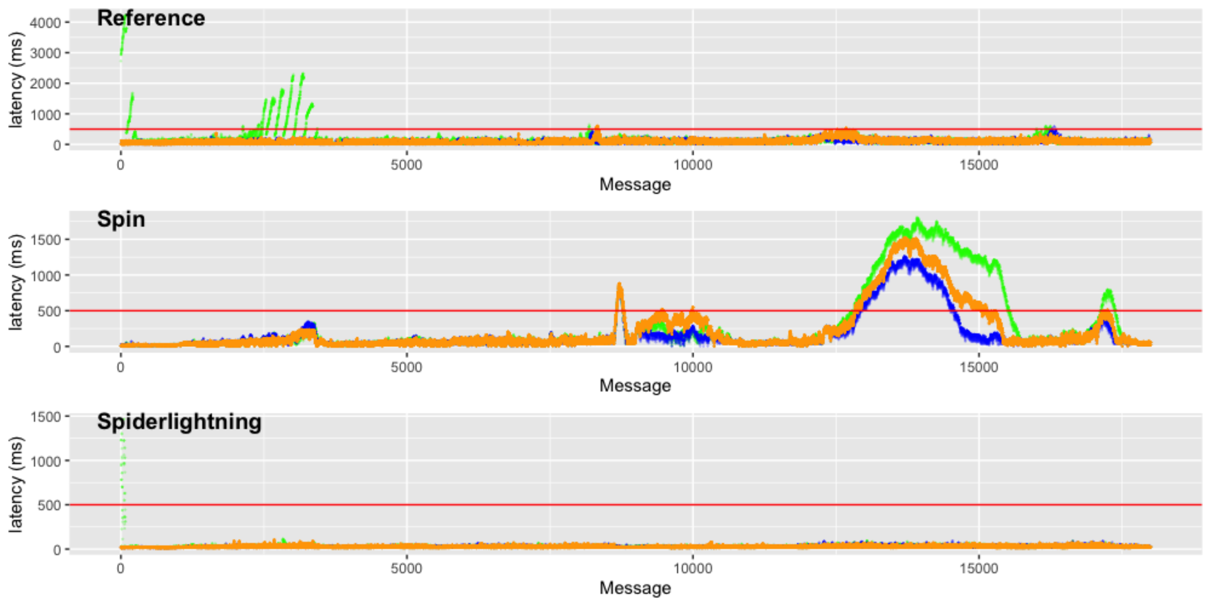


Figure 22: Evaluation with a dual core CPU, original frequency

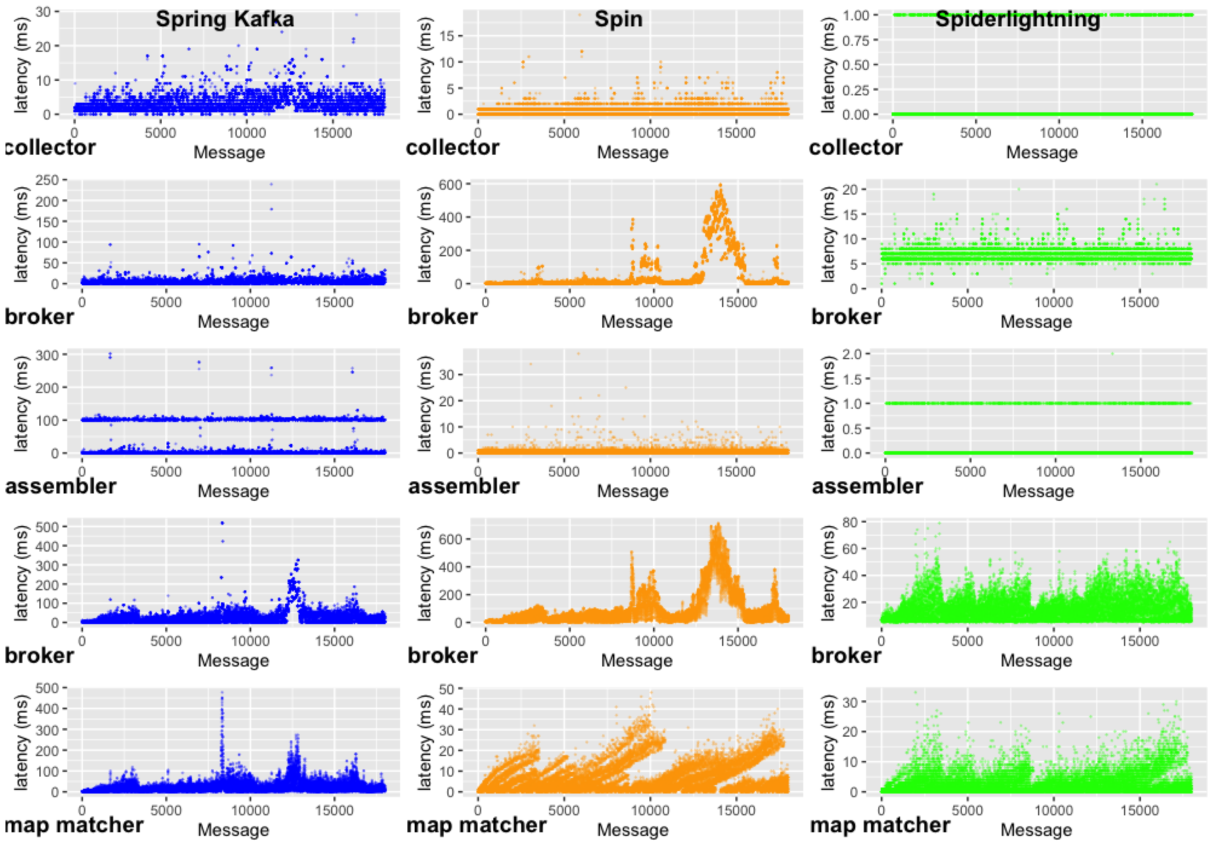


Figure 23: Dual core test run, original frequency

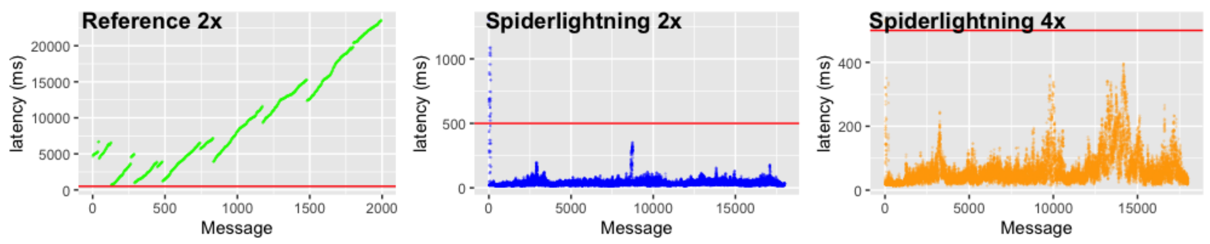


Figure 24: Dual core test run, multiplied frequency

4.5.4 Package size

In the table 9 can be seen that the size of the WebAssembly packages is significant lower than the containerized applications. The WebAssembly components are in sum between 82 and 178 times smaller than the applications from the reference implementation.

	Reference	Spin	Spiderlightning	WasmCloud
collector	354.87 MB	2.16 MB	2.16 MB	9.5 MB
assembler	380.67 MB	2.16 MB	2.16 MB	2.05 MB
map matcher	393.62 MB	2.01 MB	2.14 MB	2.08 MB
sum	**1,129.16 MB**	**6.33 MB**	**6.46 MB**	**13.63 MB**

Table 9: Package sized

5 Discussion

The evaluation shows, that WebAssembly is capable to be used within a stream data processing pipeline. Although, it depends on the used framework or runtime. Two of the three web assembly approaches were able to meet the goal, which was to ensure that the 95th percentile latency was lower than a given time. Although this goal was only reached, when the messages are processed in the actual message frequency. Furthermore, the packaging size is significant smaller for the WebAssembly approaches, which might be a benefit for some use cases.

Reference Spring Cloud Streams, as well as Kafka Streams, are both possible approaches, as the reference pipeline could manage to meet the requirement for the latency. Although, this implementation could have been more efficient, and most certainly faster, if it had been implemented as a single Kafka Streams application, where the individual steps are handled by the stream processors instead of standalone applications. This would have also resulted in smaller containers, as there would only be one JVM running instead of three. Although, as seen in the evaluations, the part which has been a bottleneck for this implementation was the map matcher, which could not keep up with the messages as the number of locations in the trajectories increased.

WasmCloud WasmCloud offers some interesting features, like the deployment of modules within a simple web UI, but also in the form of configuration files. This feature makes it easy to deploy and scale services. Furthermore, the framework handles all aspects of messaging, storage, and so on, so the developer only needs to work on the actual business logic. The abstraction does not require knowledge of the framework itself. While WasmCloud is easy to use, and building services without worrying about implementations for message brokers and other external services makes it easy to develop services, WasmCloud couldn't reach the goal for the message processing latency. An issue within the WasmCloud implementation, which might cause the slow message processing, is that the Redis key value store, which is used for the assembler service, does not work with protocol buffers. This is caused by the actual implementation of the capability, as it only accepts valid utf-8 strings as values. Therefore, the state is stored as JSON object, which may lead to massive files which are potentially received, deserialized, parsed, merged, serialized and sent to the Redis key value store again. The roundtrip for reading the geometry of the streets, which is needed for the WasmCloud implementation, is also a possible issue. With every incoming message, the map matcher calls a function on the WasmCloud host, that reads the static file from the file system. As there is, in contrast to the individual WebAssembly components, only a single host running, this might lead to the observed slow message processing. Because this implementation could not meet the requirements, even with the original message frequency and plenty of computing resources, it can be considered as not viable. Furthermore, the announcement, that there will be a major change in the way of handling the capabilities in the near future, as mentioned in 2.7.3, means that services may need to be re-written when upgrading. With this consideration, WasmCloud is not yet usable for the given use case. Once there are more implemented capabilities and the framework is using the standardized WebAssembly component model for its capabilities, it might be considered again, as the framework itself is easy to use and scale.

Spin The serverless Spin approach offers similar features as WasmCloud, but the capabilities like messaging or key value stores are not as loose coupled. This means there is no general messaging interface, but there is a dedicated component for every implemented functionality. Spin is already utilizing the WebAssembly component model, even it is not completely finalized. This means that components can be used with other frameworks, if the provided components share the same interface. Like with the WasmCloud platform, the provided functionality is abstracted away, which makes building services simpler. The scaling of the Spin components is completely managed by the Spin host and since the modules are not running if not used, they do not generate load between messages. Even though every WebAssembly module has to be initialized for every incoming message, Spin was able to meet the targeted 95th percentile latencies. When the frequency of the messages is raised, the goal is no longer achieved. The reason for this is, that the time between the processing of the messages increases considerably when the number of messages per second is increased. When running on a scaled down system, the required latencies could also not be achieved. Although, the processing time of the individual services is still lower than within the reference implementation. The issue which has been mentioned in 3.1, that the geometry has to be read from the file system on every message, does not lead to slow processing times. With this information, the bottleneck can either be the Redis pub/sub broker or the Spin runtime itself. Unfortunately, there is no other available broker yet, which can be used instead. Because of the too high latencies between the services, at higher message frequencies, Spin is also not a viable approach, as it is likely that the number of messages increase over time. Since Spin is open source, and the number of available triggers and capabilities is getting higher, the framework should be considered again, as soon as there are capabilities for streaming engines like Apache Kafka are available.

Spiderlightning/Slight The Spiderlightning runtime can be seen as a more traditional approach and is comparable with a deployment within containers. Deis Labs research, which owns Spiderlightning, also provides a way to manage those WebAssembly modules within Kubernetes clusters [51]. Within the Spiderlightning runtime, there are several capabilities like messaging via Apache Kafka or other brokers included. Those capabilities can be passed to the WebAssembly modules if they are importing a matching interface. Those capabilities are, like in the other approaches of this evaluation, abstracted away, and the developers only need to implement the business logic. Since the framework, does not add any other functionality, scaling needs to be done manually. Except for the evaluation that was using only a single CPU core, Spiderlightning managed to meet the required 95th percentile in every single evaluation. It even managed that the 99th percentile latency is below the required 500 milliseconds. It has also been shown that it performed impressively well with limited resources. This was shown in the last test case, in which the performance of the processing pipeline performed better on a striped down system than the reference pipeline with six times as much CPU cores and eight times more memory. Considering the performance of the pipeline, Spiderlightning can be seen as a viable option for a stream processing pipeline. Although, it should be noted that some workarounds have been needed, to make the implementation useable. An example for this is, that there were no capabilities available for database access or for generating HTTP calls, and there was also no capability to read files from the file system. Furthermore, as described in 3.3, the state for this specific implementation is in memory only. If a distributed state, which evicts after a given time, is needed, there is no way to implement it, with the current capabilities, within a reasonable extent. Another thing that should be noted is that since Spiderlightning is used like a traditional container, most of the benefits of using WebAssembly instead of a containerized application are not utilized. The benefits, which are still valid, are the size of the modules and the performance.

51. https://github.com/deislabs/containerd-wasm-shims visited at 22.11.22

6 Conclusion and Future Work

6.1 Conclusion

The observations show that it is possible to implement event-driven data processing, like in a streaming data processing pipeline, with the currently available WebAssembly frameworks that are utilizing WASI. All three frameworks, which have been evaluated, can be used for event-driven data processing, and have different strength and weaknesses.
While the latency with the WasmCloud implementation was too high for the evaluated use case, it is still capable for the usage of event-driven data processing. The currently available capabilities for messaging, key value stores, blob stores, and HTTP client and server cover many common use cases for the processing of a continuous stream of messages. Because of the usage of NATS as broker, it can also act as a MQTT broker, which is a common protocol for exchanging messages between different systems. Because of the web UI, which enables to add, remove and link different actors to each other, it makes the composition of event driven distributed systems straight forward. A missing piece, is a proper way of handling stateful services, although this feature has been available with a previous version of the platform, it is not anymore. WasmCloud is rewriting all its implementations for its capability providers so that they will use the component model. This should lead to a better interoperability with other WASI based systems, but will lead most certainly to breaking changes.
Spiderlightning performed exceptional well for the given use case and could outperform its peers when it comes to performance. It is not like WasmCloud or Spin, a runtime that can be used for managing and scaling WebAssembly modules. It is just a minimal core host runtime which has a set of implemented capabilities with their corresponding interfaces, which are exposed via an early implementaion of the component model. Since the Spiderlightning components are not managed by the framework, they are intended to be deployed within a Kubernetes cluster using the Containerd Wasm Shims [52] provided by Deis Labs research. This leads to an almost identical usage as with traditional containerized applications. This makes an integration or a replacement of an existing platform straightforward, but does not take advantage of the benefits that are provided by WebAssembly based services. Given, that the performance is significant better, than even the reference implementation, and the interchangeable implementations for the provided capabilities, it is definitely a recommended approach if the available capabilities met the demand. Because it uses the WebAssebly component model for its capabilities, the modules can also run within other WebAssembly frameworks if the interfaces are matching. Therefore, it might also be an appropriate framework to use, for in between, until other frameworks like Spin are implementing better support for actual stream data processing, or actual stream processing frameworks are introduced.
The Spin framework is utilizing many features of WebAssembly, which are making Wasm attractive at first place. This means, is scales from zero to as many instances as needed, as it takes advantage of the fast startup time of the WASI instances, by only starting services when they are called. It uses only standardized parts of the WebAssembly system interface, like the early implementation of the component model. Therefore, Wasm modules for Spin will also work

52. https://github.com/deislabs/containerd-wasm-shims visited at 22.11.22

within other frameworks, which are sharing the same interfaces. The issue with Spin is, that it is currently not targeting event streaming use cases like the evaluated stream processing pipeline is. Therefore, capabilities for streaming engines like Apache Kafka or Redis Streams are not implemented yet. Although, with the available capabilities, it was already possible to meet the requirements for the 95th percentile latency for the given use case. It also has been shown, that the instantiation time of the WebAssembly modules does not lead to any issues. From the evaluations with higher message rates, it is evident that the main issue leading to high latencies is not originating from the WebAssembly modules, but from the handling of threads within the Spin runtime itself, or the way the pub/sub broker is connected. This leads to the conclusion, that the serverless approach, like Spin provides, is suitable for stream processing. Although, if stateful streaming is necessary, the Redis pub/sub implementation should be replaced by a real streaming engine like Redis Streams or Apache Kafka.

6.2 Future Work

According to a survey from the Cloud Native Computing Foundation (CNCF (2022)), the interest in Wasm outside the web browser is growing fast, and already more than 30 percent of the questioned companies are using WebAssembly, while the number of companies, that want to use WebAssembly as part of their cloud infrastructure, within the next year is again over 30 percent. Furthermore, with the announcement from Irwin (2022), that Docker will support WebAssembly modules inside a container, which uses the Wasm runtime instead of a usual container runtime, the progress of Wasm should get accelerated further. There are other frameworks and platforms emerging, which would allow the implementation of event-driven data processing. Suborbital introduced E2Core[53] which usage is similar to the one of Spin, although it is not released yet. Moreover, the Layotto application runtime from MOSN[54] has introduced its support for Wasm modules as workloads. Although these approaches have not been evaluated, the rapid growth of available tools and frameworks can be considered as proof of the capability of WebAssembly outside the browser.

With the upcoming, and the already been introduced frameworks, those should be compared with the already evaluated ones, especially if they can overcome the found limitations, like the performance issue that has been introduced for the Spin implementation, when the number of messages is raised, but also in how well they can be implemented in existing systems.

The WebAssembly component model is another aspect which should be evaluated. The evaluated systems implement external functionality with this approach. It is intended to facilitate interoperability between different systems that are utilizing the component model. It will be exciting to see if the different vendors can agree for uniform interfaces, and if a Wasm module, which is used within a specific framework, can be used within other systems. As soon as the component model is completely finalized, there should be an evaluation, if such an interoperability is actual possible, and what limitations such an approach will introduce.

53. https://github.com/suborbital/e2core visited 11.11.22
54. https://github.com/mosn/layotto visited 11.11.22

Another possible research is, once the component model is finalized, to build the processing steps of the event streaming pipeline as Wasm components, that can be put together to form the stream processing pipeline, within one main application, instead of the microservices. A possible approach would be, having Wasm components that are exporting their processing functions, which are then imported from the host and applied after each other on a data stream. The interfaces for those exported functions, could be designed in a way to match the common stream processing tasks 2.2.2. This would remove the internal network requests entirely, would still use reusable pieces, and would also be easily exchangeable, as the concrete implementations for the applied processing steps are not part of the actual application. Those implementations could even be done in different programming languages.
Another future research can also evaluate, how far such interoperability can go and if it makes sense. In theory, the WebAssembly modules, used within this thesis, can be run within any web browser, or even on microcontrollers. An interesting aspect would be, to build a distributed application out of Wasm components once. Those components can then be distributed within all involved devices as needed. For example, moving parts to the edge, but if the edge computing device is only a microcontroller and can't process the data fast enough, only run a small part of the pipeline on it. Or only have a messaging client on the edge and use the entire platform as it is running in the cloud, without changing anything on the components. Or parts of the application could be executed within a browser. Here the question is, if such a distributed system would bring benefits, when compared to a cloud hosted peer.

References

Agha, Gul. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA, USA: MIT Press. ISBN: 0262010925.

Alliance, Bytecode. 2022. *wit-bindgenn,* November. https://github.com/bytecodealliance/wit-bindgen.

Apache Foundation. 2022a. *Apache Flink® — Stateful Computations over Data Streams.* Accessed: 2022-11-29. https://flink.apache.org/.

———. 2022b. *Apache Storm Documentation Version 2.4.0.* Accessed: 2022-11-29. https://storm.apache.org/releases/2.4.0/Stream-API.html.

———. 2022c. *Kafka Streams, Core Concepts.* Accessed: 2022-11-29. https://kafka.apache.org/33/documentation/streams/core-concepts.

———. 2022d. *Spark Streaming Programming Guide.* Accessed: 2022-11-29. https://spark.apache.org/docs/3.3.1/streaming-programming-guide.html#overview.

———. 2022e. *Spark Streaming Programming Guide.* Accessed: 2022-11-29. https://flink.apache.org/flink-applications.html.

Bhonsle, Archit, Ved Patil, Tanvi Valkunde, and Trupti Lotlikar. 2022. "Linear Algebra in the Browser powered by WebAssembly." In *2022 International Conference for Advancement in Technology (ICONAT),* 1–7. https://doi.org/10.1109/ICONAT53423.2022.9725939.

Butcher, Matt. 2022a. *Finicky Whiskers (pt. 4): Spin, Containers, Nomad, and Infrastructure,* June. https://www.fermyon.com/blog/finicky-whiskers-part-4-infrastructure.

———. 2022b. *The Scale to Zero Problem,* April. https://www.fermyon.com/blog/scale-to-zero-problem.

Bytecodealliance. 2022. *WebAssembly Micro Runtime,* November. https://github.com/bytecodealliance/wasm-micro-runtime.

———. *Bytecodealliance/wasmtime: A fast and secure runtime for WebAssembly.* https://github.com/bytecodealliance/wasmtime.

———. *Wasmtime documentation.* https://docs.wasmtime.dev/.

Chintapalli, Sanket, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, et al. 2016. "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming." In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW),* 1789–1792. https://doi.org/10.1109/IPDPSW.2016.138.

Clark, Lin. 2019a. *Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly,* November. https://bytecodealliance.org/articles/announcing-the-bytecode-alliance.

Clark, Lin. 2019b. *Standardizing Wasi: A system interface to run webassembly outside the web – mozilla hacks - the web developer blog,* March. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/.

Cloudflare. 2022. *Cloudflare Docs.* Accessed: 2022-11-22. https://developers.cloudflare.com/workers/learning/security-model/#freezing-a-spectre-attackl.

CNCF. 2022. *CNCF WASM microsurvey: A transformative technology, YES, but time to get serious,* November. https://www.cncf.io/blog/2022/10/24/cncf-wasm-microsurvey-a-transformative-technology-yes-but-time-to-get-serious/.

Dean, Alexander G., and Valentin Crettaz. 2019. "2.2." In *Event streams in action: Real-time event systems with Kafka and Kinesis,* 29–32. Manning Publications Co.

Deis Labs Research. 2022. *Spiderlightning.* Accessed: 2022-11-25. https://github.com/deislabs/spiderlightning/blob/main/docs/primer.md.

Dice, Joel. 2022. *The WebAssembly Component Model,* July. https://www.fermyon.com/blog/webassembly-component-model.

Dongen, Giselle van, and Dirk Van den Poel. 2020. "Evaluation of Stream Processing Frameworks." *IEEE Transactions on Parallel and Distributed Systems* 31 (8): 1845–1858. https://doi.org/10.1109/TPDS.2020.2978480.

ETSI. 2019. *ETSI EN 302 637-3 v1.3.1.* 74. April. https://www.etsi.org/deliver/etsi_en/302600_302699/30263703/01.03.01_60/en_30263703v010301p.pdf.

———. 2021. *ETSI TS 103 324 v0.0.20.* 44. February. https://portal.etsi.org/eWPM/index.html#/schedule?WKI_ID=46541.

Fallin, Chris. 2022. *Wasmtime 1.0: A look at performance,* September. https://bytecodealliance.org/articles/wasmtime-10-performance.

Fermyon Technologies. 2022a. *Extending and embedding Spin.* https://developer.fermyon.com/spin/extending-and-embedding.

———. 2022b. *Introducing Spin.* https://developer.fermyon.com/spin/index.

Gackstatter, Philipp. 2021. *A webassembly container runtime for serverless edge computing,* May. https://repositum.tuwien.at/handle/20.500.12708/17598.

Garillot, Francois. 2019. *Stream processing with Apache Spark: Mastering structured streaming and spark streaming.* O'Reilly Media, Incorporated.

Harsh, Varshney. 2022. *What is Stream Processing?* https://hevodata.com/learn/stream-processing/.

Hickey, Pat, Jakub Konka, Dan Gohman, Sam Clegg, Andrew Brown, Alex Crichton, Lin Clark, et al. 2020a. "WebAssembly/WASI: snapshot-01" (December). https://doi.org/10.5281/zenodo.4323447. https://github.com/WebAssembly/WASI.

Hickey, Pat, Jakub Konka, Dan Gohman, Sam Clegg, Andrew Brown, Alex Crichton, Lin Clark, et al. 2020b. “WebAssembly/WASI: snapshot-01” (December). https://doi.org/10.5281/zenodo.4323447. https://github.com/WebAssembly/WASI/blob/main/Contributing.md.

Hoffman, Kevin. 2022. *Using Capabilities to Decouple Non-Functional Requirements,* October. https://wasmcloud.com/blog/balancing_nfr_coupling/.

Hugo, Asmund, Brice Morin, and Karl Svantorp. 2020. “Bridging MQTT and Kafka to support C-its: A feasibility study.” *2020 21st IEEE International Conference on Mobile Data Management (MDM),* https://doi.org/10.1109/mdm48529.2020.00080.

Irwin, Michael. 2022. *Introducing the Docker+Wasm Technical Preview,* October. https://www.docker.com/blog/docker-wasm-technical-preview/.

ITU-T. 2021a. *Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER) X.691.* 62. February. https://www.itu.int/rec/T-REC-X.691-202102-I.

———. 2021b. *Information technology – ASN.1 encoding rules: XML Encoding Rules (XER) X.693.* 71. February. https://www.itu.int/rec/T-REC-X.693-202102-I.

Jankowski, Matthew, Peter Pathirana, and Sean Allen. 2015. *Storm Applied: Strategies for real-time event processing.* Simon / Schuster. ISBN: 9781617291890.

Jiaxiao, Zhou. 2022. *Introducing SpiderLightning - A Cloud System Interface with WebAssembly,* November. https://deislabs.io/posts/introducing-spiderlightning-and-slight/.

Karimov, Jeyhun, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. “Benchmarking Distributed Stream Data Processing Systems.” *2018 IEEE 34th International Conference on Data Engineering (ICDE),* https://doi.org/10.1109/icde.2018.00169.

Klabnik, Steve, and Carol Nichols. 2022. *The Rust Programming Language —Common Programming Concepts.* Accessed: 2022-11-29. https://doc.rust-lang.org/stable/book/ch03-00-common-programming-concepts.html.

Lehmann, Daniel, Johannes Kinder, and Michael Pradel. 2020. “Everything Old is New Again: Binary Security of WebAssembly.” In *29th USENIX Security Symposium (USENIX Security 20),* 217–234. USENIX Association, August. ISBN: 978-1-939133-17-5. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

Long, Ju, Hung-Ying Tai, Shen-Ta Hsieh, and Michael Juntao Yuan. 2021. “A lightweight design for serverless function as a service.” *IEEE Software* 38 (1): 75–80. https://doi.org/10.1109/ms.2020.3028991.

M. Tim, Jones. 2018. *The languages of data science,* April. https://developer.ibm.com/tutorials/ba-intro-data-science-4/.

Microsoft. 2022. *Introduction to Stream Analytics windowing functions,* September. https://learn.microsoft.com/en-us/azure/stream-analytics/stream-analytics-window-functions/.

Moffatt, Robin. 2018. *ATM Fraud Detection with Apache Kafka and KSQL,* October. https://www.confluent.io/blog/atm-fraud-detection-apache-kafka-ksql/.

Nats.io. 2021. *NATS Adaptive Deployment Architectures.* Accessed: 2022-11-22. https://docs.nats.io/nats-concepts/service_infrastructure/adaptive_edge_deployment.

Newson, Paul, and John Krumm. 2009. "Hidden Markov Map Matching through Noise and Sparseness." In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems,* 336–343. GIS '09. Seattle, Washington: Association for Computing Machinery. ISBN: 9781605586496. https://doi.org/10.1145/1653771.1653818. https://doi-org.ezproxy.fh-salzburg.ac.at/10.1145/1653771.1653818.

OpenJS Foundation. 2022. *Wasm modules.* https://nodejs.org/api/esm.html#wasm-modules.

Poggenhans, Fabian, Jan-Hendrik Pauls, Johannes Janosovits, Stefan Orf, Maximilian Naumann, Florian Kuhnt, and Matthias Mayr. 2018. "Lanelet2: A high-definition map framework for the future of automated driving." In *2018 21st International Conference on Intelligent Transportation Systems (ITSC),* 1672–1679. https://doi.org/10.1109/ITSC.2018.8569929.

Popić, Srđan, Dražen Pezer, Bojan Mrazovac, and Nikola Teslić. 2016. "Performance evaluation of using Protocol Buffers in the Internet of Things communication." In *2016 International Conference on Smart Systems and Technologies (SST),* 261–265. https://doi.org/10.1109/SST.2016.7765670.

Quddus, Mohammed A., Washington Y. Ochieng, and Robert B. Noland. 2007. "Current map-matching algorithms for transport applications: State-of-the art and future research directions." *Transportation Research Part C: Emerging Technologies* 15 (5): 312–328. ISSN: 0968-090X. https://doi.org/https://doi.org/10.1016/j.trc.2007.05.002. https://www.sciencedirect.com/science/article/pii/S0968090X07000265.

Randall, Liam. 2021. *wasmCloud Joins Cloud Native Computing Foundation as Sandbox Project,* August. https://cosmonic.com/blog/cosmonic-donates-wasmcloud-to-the-cloud-native-computing-foundation/.

Reese, Adam, and Matt Butcher. 2022. *Running Python in WebAssembly,* March. https://www.fermyon.com/blog/python-wagi.

Reisz, Wesley, and Lin Clark. 2021. *Lin Clark on the WebAssembly Component Model,* October. https://www.infoq.com/podcasts/web-assembly-component-model/.

Rossberg, Andreas. 2022. *Threads Proposal for WebAssembly,* November. https://github.com/webassembly/threads.

Sabby, Anandan, Bogoevici Marius, Bottard Eric, Fisher Mark, Gopinathan Ilayaperumal, Hillert Gunnar, Pollack Mark, et al. 2022. *Spring Cloud Stream Kafka Binder Reference Guide.* Accessed: 2022-11-29. https://docs.spring.io/spring-cloud-stream/docs/current/reference/html/spring-cloud-stream-binder-kafka.html.

Second State. 2021a. *Manage WebAssembly Apps Using Container and Kubernetes Tools.* Accessed: 2022-11-23, November. https://www.secondstate.io/articles/manage-webassembly-apps-in-wasmedge-using-docker-tools/.

Second State. 2021b. *Use WasmEdge Library in Programming Languages.* Accessed: 2022-11-23, November. https://wasmedge.org/book/en/sdk.html.

———. 2022a. *WasmEdge Proprietary Extensions.* Accessed: 2022-11-25. https://wasmedge.org/book/en/features/proprietary_extend.html.

———. 2022b. *WasmEdge Supported Platforms.* Accessed: 2022-11-25. https://wasmedge.org/book/en/features/platforms.html.

———. *Running WASM with WasmEdge CLI.* https://wasmedge.org/book/en/quick_start/run_cli.html.

Shymanskyy, Volodymyr. *Wasm3.* https://github.com/wasm3/wasm3.

Singh, Jaiteg, Saravjeet Singh, Sukhjit Singh, and Hardeep Singh. 2019. "Evaluating the performance of map matching algorithms for navigation systems: an empirical study." *Spatial Information Research* 27, no. 1 (February): 63–74. ISSN: 2366-3294. https://doi.org/10.1007/s41324-018-0214-y. https://doi.org/10.1007/s41324-018-0214-y.

Taylor, Thomas. 2022. *WebAssembly Components and wasmCloud Actors: A Glimpse of the Future,* June. https://wasmcloud.com/blog/webassembly_components_and_wasmcloud_actors_a_glimpse_of_the_future/.

Vinoski, Steve. 2007. "Reliability with Erlang." *IEEE Internet Computing* 11 (6): 79–81. https://doi.org/10.1109/MIC.2007.132.

WasmCloud. 2022a. *Capability Providers.* Accessed: 2022-12-01. https://github.com/wasmcloud/capability-providers/.

———. 2022b. *wasmCloud Host Runtime (OTP).* Accessed: 2022-12-01. https://github.com/wasmCloud/wasmcloud-otp.

Wasmer, Inc. 2022. *Wasmer Introduction.* https://docs.wasmer.io/.

WebAssembly specification. 2022. https://webassembly.github.io/spec/core/intro/introduction.html#design-goals.

Yuan, Michael. 2021. *Performance analysis for arm vs x86 cpus in the cloud,* January. https://www.infoq.com/articles/arm-vs-x86-cloud-performance/.

Zaharia, Matei, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2012. *Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing.* Technical report UCB/EECS-2012-259. EECS Department, University of California, Berkeley, December. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-259.html.

Appendices

A git-Repository

gitlab.mediacube.at

https://gitlab.mediacube.at/fhs41624/masterarbeit-stefan-eisl

B Vorlagen für Studienmaterial

C Archived Websites

https://web.archive.org/web/20221029011657/https://webassembly.github.io/spec/core/intro/introduction.html, letzter zugriff 29.10.22

https://web.archive.org/web/20211025080007/https://repositum.tuwien.at/handle/20.500.12708/17598, letzter zugriff 25.10.21

https://web.archive.org/web/20221126201211/https://github.com/bytecodealliance/wit-bindgen, letzter zugriff 26.11.22

https://web.archive.org/web/20221018051503/https://flink.apache.org/, letzter zugriff 18.10.22

https://web.archive.org/web/20220522193152/https://storm.apache.org/releases/2.4.0/Stream-API.html, letzter zugriff 22.05.22

https://web.archive.org/web/20221205030639/https://kafka.apache.org/33/documentation/streams/core-concepts, letzter zugriff 5.12.22

https://web.archive.org/web/20221205030756/https://spark.apache.org/docs/3.3.1/streaming-programming-guide.html, letzter zugriff 5.12.22

https://web.archive.org/web/20221024001230/https://flink.apache.org/flink-applications.html, letzter zugriff 24.10.22

https://web.archive.org/web/20221010122154/https://www.fermyon.com/blog/finicky-whiskers-part-4-infrastructure, letzter zugriff 10.10.22

https://web.archive.org/web/20220627125558/https://www.fermyon.com/blog/scale-to-zero-problem, letzter zugriff 27.06.22

https://web.archive.org/web/20221109154351/https://github.com/bytecodealliance/wasm-micro-runtime, letzter zugriff 9.11.22

https://web.archive.org/web/20221202042511/https://github.com/bytecodealliance/wasmtime, letzter zugriff 2.12.22

https://web.archive.org/web/20221203191222/https://docs.wasmtime.dev/, letzter zugriff 3.12.22

https://web.archive.org/web/20221122202947/https://bytecodealliance.org/articles/announcing-the-bytecode-alliance, letzter zugriff 22.11.22

https://web.archive.org/web/20221126051033/https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, letzter zugriff 26.11.22

https://web.archive.org/web/20220911183650/https://developers.cloudflare.com/workers/learning/security-model, letzter zugriff 11.9.22

https://web.archive.org/web/20221109175315/https://www.cncf.io/blog/2022/10/24/cncf-wasm-microsurvey-a-transformative-technology-yes-but-time-to-get-serious/, letzter zugriff 9.11.22

https://github.com/deislabs/spiderlightning/blob/main/docs/primer.md, letzter zugriff 20.09.22

https://web.archive.org/web/20220812075800/https://www.fermyon.com/blog/webassembly-component-model, letzter zugriff 12.8.22

https://web.archive.org/web/20220120004440/https://www.etsi.org/deliver/etsi_en/302600_302699/30263703/01.03.01_60/en_30263703v010301p.pdf, letzter zugriff 20.1.22

https://web.archive.org/web/20221026061606/https://portal.etsi.org/eWPM/index.html#/workitemslist, letzter zugriff 26.10.22

https://web.archive.org/web/20221123083511/https://bytecodealliance.org/articles/wasmtime-10-performance, letzter zugriff 23.11.22

https://web.archive.org/web/20221205032126/https://developer.fermyon.com/spin/extending-and-embedding, letzter zugriff 5.12.22

https://web.archive.org/web/20221109160012/https://developer.fermyon.com/spin/index, letzter zugriff 9.11.22

https://web.archive.org/web/20211025080007/https://repositum.tuwien.at/handle/20.500.12708/17598, letzter zugriff 25.10.22

https://web.archive.org/web/20221201104247/https://github.com/WebAssembly/WASI, letzter zugriff 1.12.22

https://web.archive.org/web/20220522082451/https://github.com/WebAssembly/WASI/blob/main/Contributing.md, letzter zugriff 22.5.22

https://web.archive.org/web/20221005161811/https://wasmcloud.com/blog/balancing_nfr_coupling/, letzter zugriff 5.10.22

https://web.archive.org/web/20221123142436/https://www.docker.com/blog/docker-wasm-technical-preview/, letzter zugriff 23.11.22

https://web.archive.org/web/20221205032556/https://www.itu.int/rec/T-REC-X.691-202102-I, letzter zugriff 5.12.22

https://web.archive.org/web/20221205032703/https://www.itu.int/rec/T-REC-X.693-202102-I, letzter zugriff 5.12.22

https://web.archive.org/web/20221123142446/https://deislabs.io/posts/introducing-spiderlightning-and-slight/, letzter zugriff 23.11.22

https://web.archive.org/web/20221128180242/https://doc.rust-lang.org/stable/book/ch03-00-common-programming-concepts.html, letzter zugriff 28.11.22

https://web.archive.org/web/20220826230814/https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann, letzter zugriff 26.8.22

https://web.archive.org/web/20221012035600/https://developer.ibm.com/tutorials/ba-intro-data-science-4/, letzter zugriff 12.10.22

https://web.archive.org/web/20221118082456/https://learn.microsoft.com/en-us/azure/stream-analytics/stream-analytics-window-functions, letzter zugriff 18.11.22

https://web.archive.org/web/20221023153452/https://www.confluent.io/blog/atm-fraud-detection-apache-kafka-ksql/, letzter zugriff 23.11.22

https://web.archive.org/web/20220517145123/https://docs.nats.io/nats-concepts/service_infrastructure/adaptive_edge_deployment, letzter zugriff 17.5.22

https://web.archive.org/web/20221202134431/https://nodejs.org/api/esm.html, letzter zugriff 2.12.22

https://web.archive.org/web/20220524213849/https://cosmonic.com/blog/cosmonic-donates-wasmcloud-to-the-cloud-native-computing-foundation/, letzter zugriff 24.15.22

https://web.archive.org/web/20220616062314/https://www.fermyon.com/blog/python-wagi, letzter zugriff 16.6.22

https://web.archive.org/web/20220625195801/https://www.infoq.com/podcasts/web-assembly-component-model/, letzter zugriff 25.6.22

https://web.archive.org/web/20220922111435/https://github.com/webassembly/threads, letzter zugriff 22.9.22

https://web.archive.org/web/20220627053629/https://docs.spring.io/spring-cloud-stream/docs/current/reference/html/spring-cloud-stream-binder-kafka.html, letzter zugriff 27.6.22

https://web.archive.org/web/20221007225455/https://www.secondstate.io/articles/manage-webassembly-apps-in-wasmedge-using-docker-tools/, letzter zugriff 7.10.22

https://web.archive.org/web/20221129200951/https://wasmedge.org/book/en/sdk.html, letzter zugriff 29.11.22

https://web.archive.org/web/20221129200841/https://wasmedge.org/book/en/features/proprietary_extend.html, letzter zugriff 29.11.22

https://web.archive.org/web/20221129201009/https://wasmedge.org/book/en/features/platforms.html, letzter zugriff 29.11.22

https://web.archive.org/web/20221129201009/https://wasmedge.org/book/en/quick_start/run_cli.html, letzter zugriff 29.11.22

https://web.archive.org/web/20221104071133/https://github.com/wasm3/wasm3, letzter zugriff 4.11.22

https://wasmcloud.com/blog/webassembly_components_and_wasmcloud_actors_a_glimpse_of_the_future/, letzter zugriff 16.6.22

https://web.archive.org/web/20221024201319/https://github.com/wasmCloud/capability-providers, letzter zugriff 24.11.22

https://web.archive.org/web/20220831145047/https://github.com/wasmCloud/wasmcloud-otp, letzter zugriff 31.8.22

https://web.archive.org/web/20221204180431/https://docs.wasmer.io/, letzter zugriff 4.12.22

https://web.archive.org/web/20221029011657/https://webassembly.github.io/spec/core/intro/introduction.html, letzter zugriff 29.10.22

https://web.archive.org/web/20220809161618/https://www.infoq.com/articles/arm-vs-x86-cloud-performance/, letzter zugriff 9.8.22