Goto table of contents

Duale Hochschule Baden-Württemberg Mannheim

Exploring WebAssembly for versatile plugin systems through the example of a text editor

im Studiengang TINF22IT1

an der *Duale Hochschule Baden-Württemberg Mannheim*

von

Name, Vorname:	Hartung, Florian
Abgabedatum:	15.04.2025

Bearbeitungszeitraum:	15.10.2024 - 15.04.2024
Matrikelnummer, Kurs:	6622800, TINF22IT1
Wiss. Betreuer*in der Dualen Hochschule:	Gerhards, Holger, Prof. Dr.

Erklärung zur Eigenleistung

Ich versichere hiermit, dass ich meine Projektarbeit mit dem

THEMA
Exploring WebAssembly for versatile plugin systems through the example of a text editor

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum

Unterschrift

Abstract

Plugin systems are a software architecture pattern for making host applications extensible without modification of the host application itself. They are used in various applications such as realtime audio processing or computer games, however they play an especially important role for text editors. Text editors and IDEs are applications used mostly for software development by developers. Depending on the required technologies and highly individual developer preferences, text editors must be able to adapt to a variety of different use cases. Such use cases might include language support, complex keybindings or development on a remote server.

WebAssembly (Wasm) is a relatively new technology first released in March 2017. It was originally designed as a fast, safe and portable compilation target for higher level languages to allow fast code execution on the web. However it is designed without assumptions about its execution environment and its properties make it an interesting technology for non-web contexts such as avionics or the automotive industry as well.

This work explores Wasm as a technology for implementing a safe and fast plugin system with portable plugins compiled from higher level languages, such as C, C++, Rust, Python or JavaScript. It asks the question if Wasm is able to surpass existing plugin system technologies in terms of performance, safety and its host system impact as a new versatile technology.

For that a technology comparison between selected existing plugin system is conducted to analyze the state of the art of plugin system technologies. Then WebAssembly is evaluated across the same criteria to enable an objective comparison with existing technologies. Finally as a proof of concept, a plugin system based on WebAssembly is developed for an existing text editor project, to provide better insight into the current state of the rapidly evolving Wasm ecosystem.

TODO: present results

TODO: discuss results and impact of this work briefly

Zusammenfassung

TODO: Abstract übersetzen

Contents

List of figures

List of tables

1 Introduction

1.1 Motivation & problem statement

1.2 Research question

Is WebAssembly the best technology choice for designing versatile plugin systems for text editors?

1.3 Methodology

2 Fundamentals

This section introduces theoretical and technical fundamentals used in this work. First it covers the definition of an instruction set architecture. Then it gives an overview over WebAssembly, its features, challenges, limitations and extensions. Finally, plugin systems are explained as a software architecture model.

2.1 Instruction set architectures

In his book "Structured Computer Organization" Tanenbaum defines an instruction set architecture (ISA) as a level in a multilayered computer system[1, sec. 1.1.2]. The ISA level defines a machine language with a fixed set of instructions. According to Tanenbaum the ISA level then acts as a common interface between the hardware and software. This allows software in the form of ISA instructions to manipulate the hardware. Software written in a higher level machine language (Assembly, C, Java, …) can not be executed directly by the hardware. Instead higher level machine codes are compiled to ISA machine code or interpreted by a program, that is present in ISA machine code itself [1, sec. 1.1.2].

2.2 WebAssembly

2.2.1 Overview

WebAssembly (Wasm) is a stack-based ISA for a portable, efficient and safe code format. Originally it was designed by engineers from the four major vendors to enable high-performance code execution on the web [2]. However it is also becoming increasingly interesting for researchers and developers in non-web contexts. Some examples are avionics for Wasm's safe and deterministic execution [3], distributed computing for its portability and migratability [4] or embedded systems for its portability and safety [5].

What is special about Wasm is that it is a *virtual* ISA [6, sec. 1.1.2]. There is no agreed-upon definition for a virtual ISA, however the term *virtual* can be assumed to refer to an ISA that is running in a virtualized environment on a higher level in a multilevel computer[1]. We call this virtualized environment the **host environment**

[1]Projects that try to execute Wasm directly exist. One example is the discontinued `wasmachine` project, which tried executing Wasm on FPGAs: https://github.com/piranna/wasmachine

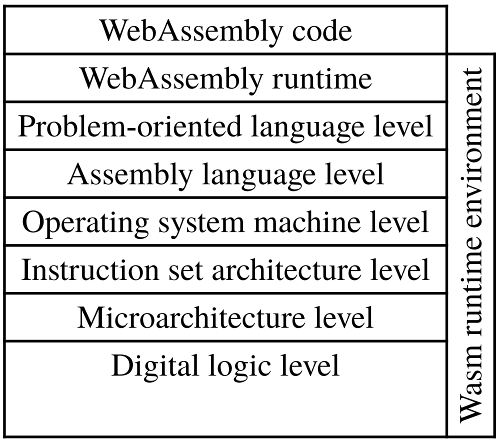

Figure 1: A multilevel computer system running Wasm code. Based on [1, fig. 1-2]

(used by the specification[6, sec. 1.2.1]) or the **WebAssembly runtime** (used by most technical documentation).

If one considers a system running Wasm code as a multi-level computer system, the Wasm runtime can be modeled as a separate layer. Figure 1 shows a multi-level computer system based on Tanenbaum's model [1, sec. 1.1.2]. Here each level is executed by logic implemented in the next lower level either through compilation or interpretation. The digital logic level itself only exists in the form of individual gates, consisting of transistors and tracks on the processors' chip. This level runs the next microarchitecture and ISA levels, which are also often implemented directly in hardware. The ISA level then provides a fixed set of instructions for higher levels to use. Operating systems build on top of this and provide another level for user space programs, which exist on the assembly language level. Then there are problem-oriented languages such as C, C++ or Rust, which are specifically made for humans to write code in[1, sec. 1.1.2].

One program written in a problem-oriented language is the Wasm runtime, which itself is a layer here. Its task it to interpret or compile higher-level Wasm code to lower-level problem-oriented or even the assembly language level. However for this work all layers starting with the Wasm runtime level until the digital logic level can be seen as a single hardware specific layer called the *Wasm runtime environment*.

WebAssembly code can be written by hand, just as one could write traditional ISA instructions (think of writing x86 machine code) by hand. Even though it is possible, it would not be efficient to write useful software systems in Wasm because the language is too simple. For example it provides only a handful of types: Signed/unsigned integers,

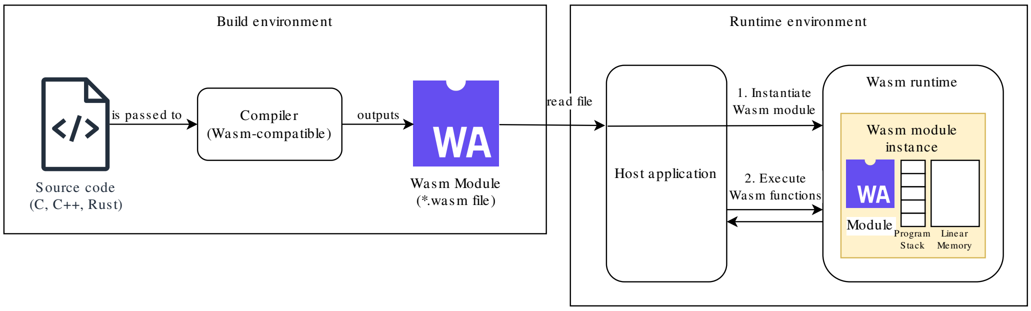

Figure 2: Flowchart for the creation of a Wasm module from a higher-level language and execution through a host application.

floating point numbers, a 128-bit vectorized number and references to functions/ host objects.

WebAssembly also does not provide any way to specify memory layouts as it can be done in higher-level languages with structs, classes, records, etc. Instead it provides most basic features and instructions, which exist on almost all modern computer architectures like integer and floating point arithmetic, memory operations or simple control flow constructs. This is by design, as WebAssembly is more of a compilation target for higher level languages[2]. Those higher level languages can then build upon Wasm's basic types and instructions and implement their own abstractions like memory layouts or control flow constructs on top. This is analogous to the non-virtual ISA machine code in a conventional computer, which also acts as a compilation target for most low-level languages such as Assembly, C, or Rust. Nowadays there are compilers for most popular languages already. Some of which are `clang` for C/C++, `rustc` for Rust or the official Zig and Haskell compilers. However most compilers are still being actively worked on and improved over time to support the latest Wasm proposals.

2.2.2 Execution model and lifecycle

Figure 2 shows the different stages a Wasm module goes through in its lifecycle. Its lifecycle starts with a developer writing source code. This source code can be written in an arbitrary programming language such as C, C++ or Rust. These languages are often used because they have compilers that support Wasm. The compiler then compiles this source code to a *Wasm module*. This step requires a compiler that support Wasm as a

[2]https://webassembly.org/

compilation target. Many modern compilers such as `clang` or `rustc` use the LLVM[3] project for optimization and code generation. These compilers compile source code to LLVM's intermediate representation (LLVM IR) and pass this to LLVM. LLVM can then perform optimizations and compile this IR to any compilation target, which can be selected by choosing a LLVM backend. One such LLVM backend targets Wasm. This way the Wasm-specific compiler logic can be implemented once within LLVM, and any compiler using LLVM can then target Wasm with minimal additional effort.

The compiled Wasm module exists in the form of a `.wasm` file. It is fully self-contained and unlike binaries it cannot depend on any dynamic libraries at runtime. It consists of different ordered sections, each with their own purpose: Some example sections contain function signatures, data segments, import- and export definition or actual Wasm instructions for each function.

The previously generated Wasm module can then be transferred to any target device or platform providing a Wasm runtime. Wasm runtimes are usually hardware-specific and not portable, unlike the Wasm module. The Wasm runtime is able to parse the received Wasm module file, instantiate a *Wasm instance* from it and provide an Application Programming Interface (API) for interaction with the Wasm instance. APIs can differ from one Wasm runtime to another. Some runtimes exist as standalone programs that can run Wasm modules comparable to how native binaries can be executed. Others like the one shown in Figure 2 are in the form of libraries, that can only be used from a host application to embed a Wasm runtime into them. These Wasm runtime libraries often provide common operations to the host application like calling Wasm functions, reading and writing operations for Wasm memories, linking mechanisms between Wasm modules, exposing host-defined functions for Wasm instances to call, etc.

In a web context a server might provide this Wasm module to the client's browser, which usually comes with a Wasm runtime. Detailed information on the current status of Wasm support for internet browsers can be viewed at https://caniuse.com/wasm. A concrete example is Ebay using Wasm for their barcode scanner algorithm to achieve higher performance[4].

---

[3] According to its official website the acronym *LLVM* was once short for *Low Level Virtual Machine*, however now it has "grown to be an umbrella project" referred to simply as the LLVM project. https://llvm.org/

[4] https://innovation.ebayinc.com/tech/engineering/webassembly-at-ebay-a-real-world-use-case/

For distributed computing and especially serverless functions a compiled Wasm module could be distributed among multiple different nodes regardless of their platform and architecture while still providing good performance. This is what the projects WasmEdge or Fermyon (see https://wasmedge.org/ and https://www.fermyon.com/) are trying to achieve.

2.2.3 Design goals

Wasm was designed with certain design goals in mind. This section presents the design goals relevant for this work according to the official Wasm specification [6, sec. 1.1.1]. Each design goal is accompanied by related information from various papers and articles for a deeper understanding of each goal.

2.2.3.A Fast

Wasm is designed to be fast both during startup and execution [6, sec. 1.1.1]. Startup time is mostly optimized through the structure of the Wasm bytecode format, which is optimized for fast parsing and compiling of Wasm code. To further elaborate, a Wasm module in its binary format consists of 11 different sections. These sections must only occur in a very specific order to allow Wasm runtimes to parse and/or skip them efficiently. Another example is the absence of backwards jump (except for the `loop` statement), which allows the use of faster one-pass compilers.

During runtime Wasm's execution time varies. The original design paper for Wasm found Wasm to be at most two times as slow as native code and around 30% of PolyBenchC's benchmarks to close to native code execution time [2, p. 197]. Another performance analysis by Jangda et al. measures peak slowdowns of 2.5x (Chrome) and 1.45x (Firefox) for running Wasm code in different browsers and comparing their execution times to native execution[7].

An additional property of Wasm bytecode is that it can be either compiled, interpreted or just-in-time compiled by a runtime. This provides users of Wasm with flexibility for performance optimizations. Some use cases require the fast startup time, determinism and relocatability (see Section 2.2.3.F) only an interpreter can provide. Others require high performance at the cost of flexibility of a compiler or just-in-time compiler.

2.2.3.B Safe

Wasm's execution is safe and sandboxed by default. All operations operate on a stack, which contains values, labels as references for branch instructions and activations, which are similar to function call frames. The Wasm stack differs from a stack used in most native architectures in two ways: Wasm code can only access the top-most value of the stack, and the stack is type-checked by the Wasm runtime during a validation phase[6].

Wasm modules also have their own linear memory. This linear memory is completely isolated from the stack and can only be accessed using special instructions and bounds-checked indices instead of pointers. While host applications can always read from and write to Wasm linear memory, Wasm instances cannot access host memory in the opposite direction. Instead, Wasm instances can only interact with the host through functions explicitly exposed by the host[6].

While the Wasm execution format is sandboxed, it is still important that the implementing Wasm runtime does not introduce any sandbox escapes due to bugs, etc. In their article "Provably-Safe Multilingual Software Sandboxing using WebAssembly" Bosamiya et al. have acknowledge Wasm as a safe sandboxed execution format and implemented a provably safe and verifiable Wasm runtime[8]. This shows that implementation of a fully sandboxed and safe runtime is possible not only in theory but also in practice.

Because Wasm is able to provide safety for untrusted code execution, it is currently gaining interest in certain safety-critical non-web contexts, such as avionics[3] or automotive industries (see https://oxidos.io/).

2.2.3.C Portable

Originally Wasm was designed for fast and safe code execution on a client's web browser[2]. This means that a variety of web browsers, all running on different platforms and architectures from desktop computers, mobile devices and even embedded devices, have to be able to run Wasm code.

Thus Wasm was designed to be able to be portable for various target platforms and architectures[6]. It does this by defining only the most basic types and instructions necessary to act as a compilation target. The basic types and instructions chosen exist on most platforms and architectures today. For example Wasm defines ~172 different instructions and ~8 types[6, sec. 4.2.1]. In comparison, the x64 architecture defines ca. 1500 - 6000 instructions[8].

2.2.3.D Independence of language

Wasm is designed to be a compilation target for higher level languages. In particular it is not designed for a specific language, programming model or object model[6, sec. 1.1.1]. Currently Wasm can already be used as a compilation target by various languages such as C, C++, Rust, Haskell or Ada.

2.2.3.E Compact

The representation of the Wasm bytecode format is designed to be compact. Especially on the web file size is very important to minimize loading times and achieve better user experiences. Also smaller files are faster to load into memory, which might lead to a slight increase in performance.

Research shows that the compactness of Wasm bytecode in non-web contexts is close to native code with around 246% the size of natively compiled x86_64 machine code for the PolyBenchC benchmarks[9]. While Wasm should optimally come as close to native code compactness as possible, this is close to impossible as native code is able to provide a larger variety of instructions exposing hardware-specific behavior.

As a concrete examples for how Wasm bytecode is compacted, variable integers are explained in the following: Often times, the top-most bytes of integers are 0. To reduce the size of integers, their bit information is not stored as a fixed amount of bytes (e.g. 4 bytes for a 32-bit integer) anymore. Instead only 7 bits per byte are used to store the integer's bits, while the 8th bit is reserved for a flag that indicates whether the next byte also belongs to the integer currently being encoded. This allows to store 32-bit unsigned integers in the interval $[0; 127]$ inside one byte, $[128; 16383]$ inside two bytes, etc., while the maximum possible integer value of $2^{32} - 1$ now requires 5 bytes for storage. This design is very similar to how Unicode encodes multi-byte characters.

2.2.3.F Other features

This section lists some other noteworthy features and design goals of Wasm. These are not directly related to this work, but instead provide a better overview over Wasm's potential use cases and applications.

- **Determinism**: The majority of Wasm code is deterministic and undefined behavior is prevented through validation before execution of Wasm code happens. There are only three possible sources of indeterminism:
  - Functions exposed by the host application and imported into a Wasm instance can cause side-effects and introduce indeterminism[6, sec. 4.4.10].
  - NaN float values and operations on them are not defined deterministically, because there exists a variety of different NaN values[6, sec. 2.2.3].
  - Wasm instances can grow their memory or tables through the `memory.grow` and `table.grow` instructions. Wasm runtimes are able to make these instructions fail, either due to missing resources, because of resource limitations or any other reason without determinism[6, sec. 4.4.6, sec. 4.4.7].
- **Parallelizable**: Wasm bytecode is designed for operations on it to be easy to parallelize[6, sec. 1.1.1]. This applies to decoding, validation and compilation of Wasm bytecode and it may allow for faster startup times of Wasm instances.
- **Streamable**: With Wasm being made for the web, the streamable property is quite unique from other ISAs. This property means that Wasm bytecode is able to be read and consumed by a Wasm runtime, before the entire data has been seen[6, sec. 1.1.1]. On the web this means, that the Wasm runtime inside a client's browser can start parsing and validating a Wasm module while it is being received over the internet. This reduces startup time for Wasm instances especially on the web, but it may also be relevant for non-web contexts, e.g. where large Wasm modules are being read from slow hard disk drives.
- **Modular**: Wasm applications consist of smaller modules, so called Wasm modules. They are self-contained, which allows them to be "transmitted, cached and consumed separably"[6, sec. 1.1.1]. During runtime most Wasm runtimes also allow multiple Wasm modules to be linked together, as long as all imports required by those Wasm modules are fulfilled.
- **Well-defined**: Wasm code is designed to be easy to understand and "reason about informally and formally"[6, sec. 1.1.1] without loopholes or undefined behavior.
- **Relocatability**: Wasm modules by themselves are clearly defined at runtime consisting of their bytecode, a linear memory, reference tables, etc. Thus it is possible to halt execution of Wasm instances, serialize the entire runtime state and relocate it to another computer system for continuation of execution in theory. This could enable

```
char* get_c_string() {
    ...
}

fn get_rust_string() -> String {
    ...
}
```

(a) Functions signatures in the C and Rust programming languages.

```
// C
(type (result i32))

// Rust
(type (result i32) (result i32))
```

(b) WebAssembly functions in its text format for the C and Rust programs

Figure 3: Function signatures in the C and Rust programming languages and their compiled type signatures in the WebAssembly text format after compilation. Both functions take no arguments and return a string.

new methods for debugging, where traditionally a runtime state and logs are dumped for later analysis by a developer. Now an entire snapshot of the runtime state could be created, that can be rerun and analyzed at a later time.

2.2.4 Challenges & Limitations

This section deals with common challenges and limitations of Wasm especially important in non-web contexts.

Wasm provides only a very basic type system with 8 types: 32-bit and 64-bit integers and floats, a 128-bit vector type and three types of references to functions, host objects and the null reference. Its type system does not contain any methods to combine multiple data types into a new one, such as structs or classes which are common in most programming languages. This is intentional, so that Wasm can be a universal compilation target independent of the higher-level language used (see Section 2.2.3.D), because not all languages share the same data constructs. However this basic type system also has its downsides. Because Wasm delegates the responsibility of managing data layouts and representations to compilers, interfaces are now depending on the specific compiler used to create the Wasm module. Consider as an example the different strings representations in C and Rust. Traditionally strings in C exist as pointers to null-terminated byte sequences of ASCII characters. In Rust strings use the Unicode format by default. Rust strings still store a pointer to byte sequence, however characters are encoded, allowing for multi-byte characters for example. Also Rust strings do not use null-terminators, and instead store the length of the string alongside the pointer in a `String` struct.

Now if one were to compile a C program exposing a function returning a string and a Rust program exposing a function that also returns a string, these would compile to different Wasm function types.

Figure 3a shows two function signatures in the C and Rust programming languages for functions returning a string. When these functions are compiled to WebAssembly, they result in two different function signatures shown in Figure 3b. The compiled C function returns a pointer in the form of an integer to a null-terminated string. The compiled Rust function returns two integers, one as the pointer to the bytes of a Unicode string and the other as the length of the string in bytes.

This issue exists for the various types commonly found in programming languages, for example consider the data layout of enums, tagged unions, futures, options or results. There are multiple solutions, for how Wasm interfaces can be standardized across programming languages:

1. A common serialization format such as JSON, XML or a binary format such as CBOR can be used. Before the host invokes a Wasm function, it has to serialize all arguments into a large byte array. Then this byte array passed indirectly as an argument, by writing it into the Wasm instance's linear memory and passing an index to the data to the function. Internally the function first has to deserialize the data into its original form, before it can execute its program logic. At the end of the function, the same procedure is repeated to pass return values back to the host. For this approach a library such as Protobuf (see https://protobuf.dev/) could also be used. It provides serialization and deserialization logic for a variety of programming languages. However this approach might introduce both a performance and memory overhead, due to the heavy serialization and deserialization necessary.
2. Another approach is to specify a custom Wasm interface for every type and function. This is more labour-intensive, as it requires developers to write glue code for every language and function by hand.

   One project that builds upon this approach is the WebAssembly Component Model presented in Section 2.2.5. It provides a new language for specifying interfaces and can generate glue code for supported programming languages such as Rust, C, C++, Python, JavaScript, etc. Even though it may look similar to the first solution of using standardized serialization, this approach is specifically tailored to Wasm. Thus is can achieve better performance and efficiency.

Another related challenge are common features of the operating system used by almost all programming languages such as file systems, networking or RNG. Most standard libraries rely on these functions to provide abstractions, e.g. over reading data from a file. An interface for these common APIs must be defined, that is language-agnostic, independent of compilers and compatible across all host platforms, on which the Wasm runtime will eventually execute the Wasm code. A solution generally accepted by compilers and standard libraries is the WebAssembly System Interface (WASI) presented in Section 2.2.6. It also relies on the WebAssembly Component Model for language-agnostic interface definitions.

2.2.5 WebAssembly Component Model

TODO

2.2.6 WebAssembly System Interface (WASI)

TODO

2.3 Plugin systems

TODO

3 Technology comparison of existing plugin systems

To evaluate whether Wasm is a viable technology for versatile plugin systems, one must first understand what criteria make a plugin system good and versatile. This section will perform a technology comparison between several technologies and existing software projects. First, a set of criteria is defined. Then, appropriate technologies and software projects are selected to represent a spectrum of different plugin systems. Next, the technologies and projects are evaluated against the previously defined criteria. Finally, a technology comparison matrix is used to summarize and visualize the results.

3.1 Definition of criteria

In this section criteria for good plugin systems are defined. Each criterion will define a scale from 0 to 5, along with requirements for each score. This scale will be useful later to enable an objective evaluation and comparison of technologies and projects.

3.1.1 Performance

A computer's performance usually refers to the speed it is able to execute software at. For interactive computer systems one generally wants every piece of code to run as fast as possible to minimize its time on the CPU.

In the context of plugin systems, performance also refers to the speed at which software is executed. The three relevant software components necessary to define performance of a plugin system technology are the host system, the plugin system and the plugins managed and called by the plugin system. For this work we define performance as the property that describes how quickly a host system can temporarily transfer execution to a plugin system, which then loads and invokes a plugin's function.

While performance can be measured quantitatively through benchmarks, in practice this is quite hard for plugin systems. Plugin systems and their technologies often vary between host applications as they are by nature highly individual. To benchmark different plugin systems one would have to implement a variety of algorithms and scenarios for a variety of plugin systems. Then one could measure the time each plugin system and plugin takes to execute.

Due to time-constraints and the broad spectrum of knowledge about programming languages and host applications necessary, this work does not use quantitative benchmarks to measure performance. Instead performance is judged through educated guesses based on benchmarks and comparisons already available for the technologies chosen and built upon by plugin systems.

0 – Very slow The transfer of execution to the plugin systems and invocation of a plugin is highly inefficient. Thus the plugin system is not viable for use within interactive software. Reasons for significant bottlenecks might include heavy serialization or expensive VM-based sandboxing.

1, – Slow Both the plugin system and plugins run very slowly. Transferring execution between the host system and a plugin is inefficient. Therefore this plugin system is also not recommended for use in interactive software, unless these inefficiencies can be somewhat mitigated, e.g. by offloading to other threads.

2, 3 – Acceptable The plugin system and plugins are not fast but their performance is acceptable for interactive software such as text editors. They can negatively impact the user experience by causing stuttering or slow loading times, but there are workarounds to minimize the impact of these problems.

4 – Fast Transferring execution to the plugin system and/or executing a plugin is fast, with only a small overhead, not noticeable by a user. While there is still a small overhead present, it is usually negligible in practice, except in scenarios with real-time requirements.

5 – Optimal Transferring execution and invoking a plugin is virtually instantaneous. There is no measurable overhead. All plugin code executes as fast as if it were implemented natively within the host system.

The scoring outline presented here is intentionally not very specific, without any hard lines between the different scores. It is meant to give only a rough guideline for evaluation, which then needs to be done very carefully on a case-by-case basis. For example, one could evaluate plugin systems based on whether plugins are compiled/interpreted or how large and thus slow plugins might be to load.

3.1.2 Plugin size

The plugin size property refers to the average size of a plugin for a specific plugin system technology. This property does not refer to the size of one specific plugin, but rather it is used to compare different plugin system technologies and how compact and small plugins for them are generally. The average plugin size may vary from technology to technology due to factors such as static vs. dynamic linking of libraries or the size of the specific language's standard library.

The importance of plugin size depends on the specific use case and user requirements. For text editors specifically a smaller plugin size might result in faster startup times and less time spent downloading or updating the plugin. However reducing the overall impact a program has on the system's resource usage is generally preferred.

Also note that this section only refers to the plugin size and not the size of the entire plugin system. While the plugin system's size is also very important for the memory impact of the host application, it is harder to measure. This is due to the fact that plugin systems are usually very tightly coupled with the host application. To measure a plugin system's size, one would have to disable the plugin system of some host application, without breaking the host application itself. Only then would it be possible to compare system resource usages between the host application with and without a plugin system. Due to the high complexity, the plugin system's size will not be taken into account in this work.

The following scores will be used to evaluate a plugin's size. They are chosen specifically for plugin systems for text editors.

5 – Minimal(< 5KB) Plugin sizes are as minimal as they can possibly be. Plugins contain the minimal amount of program code necessary to achieve their desired functionality. The plugin code format is also made to be very space-efficient, which could be implemented for example through compression and hacks on the byte/bit levels.

4 – Negligible(< 500KB) Plugin sizes are so small that they are negligible in practice. There is no replication of similar information between multiple plugins such as statically-linked libraries.

3,2 – Moderate(< 50MB) Plugins are not very small, however their size is still quite manageable in the context of text editors. Examples could be plugin system technologies, that require a large fully-self contained runtime to be shipped with every plugin. Plugins might also have to contain all libraries and standard libraries that they depend on.

1 – Large(≤ 500MB) Plugins are unusually large specifically in the context of text editors. They are not as easy to manage and during runtime they might also have a non-negligible impact on the memory footprint of their host application.

0 – Very large(> 500MB) Plugins are very large. This may be due to the fact that their internal program logic requires very costly operations such as the virtualization of an entire environment, that must be completely self-contained in the plugin code.

3.1.3 Plugin isolation

Often times plugins contain foreign code. This is especially true for text editors, where plugins are often downloaded from a central registry, also known as plugin/extension marketplaces. This means that plugins downloaded from such sources are usually not validated and thus should be treated as foreign code. Even though there might be checks in place for malicious contents, foreign code should not be trusted to not access its host environment unless otherwise allowed.

For this work we define the property of plugin isolation to describe how isolated a plugin's environment is from its host environment. While there has to be some kind of interface between both environments to make plugins accessible and usable, this interface must not be considered when evaluating plugin isolation. Instead we define plugin isolation completely separated from the interface, meaning if an interface is unsafe by nature, plugin isolation is not automatically violated.

0 – No isolation, required elevated privileges Plugins are not only not isolated from the host application, they also require certain elevated privileges, which the host application usually does not require by itself.
Worst case: Elevated privilege access to current system.

1 – No isolation Plugins are not isolated from the host application. They inherit the host application's privileges without any attempt of the host plugin system to restrict these permissions.
Worst case: Plugins gain the same privileges as the host system, usually this means access to the current user's system and peripherals.

2, 3 – Restricted isolation Plugins are not isolated from the host application by design. Normally the plugin would inherit the host application's privileges, however the host application makes an attempt to restrict plugins from accessing certain critical functionalities. Some examples for restrictions on Linux systems could be allowing only a specific subset of syscalls through `seccomp(2)`[5] or using

[5]https://www.man7.org/linux/man-pages/man2/seccomp.2.html

`namespaces(7)`[6] to isolate and limit resources. However both of these examples do not use a sandboxing strategy for isolation.

Because these restrictions can come in a various shapes and forms each based on different technologies, during evaluation either 2 or 3 can be chosen as a score.
Worst case: Plugins have similar privileges to that of the host application, except for those specifically disallowed. However this restriction might be able to be circumvented.

4 – Fully sandboxed, dynamic interface Plugins generally run completely isolated. However their interface is not statically defined, which can lead to vulnerabilities of the interface during runtime due to higher complexity and risk of bugs. Imagine a scenario where a host application exposes only a single interface for passing serialized messages back and forth with a plugin. Then the host application has to serialize and deserialize those messages during runtime. For complex systems, where advanced concepts such as additional shared memory between the host and plugins are used, this interface can become susceptible to logic bugs due to the dynamic interface.
Worst case: Access to parts of the host application not meant to be exposed due to a bug in the interface.

5 – Fully sandboxed, static interface The plugin runs fully sandboxed. It has no way of interacting with the host system, except for statically checked interfaces. Here statically checked interfaces refers to interfaces, that can be proven safe during compilation (or alternatively development) of the plugin system. One way to achieve this might be an interface definition in a common interface definition language. This restriction was chosen because it disallows plugin systems giving full access to parts of a host application without a proper interface definition.
Worst case: Indeterminable, a major bug in the sandboxing mechanism is required.

3.1.4 Plugin portability

Portability stems from the field of distributed systems. A high portability refers to software components, that can be moved from one distributed system A to another distributed system B without having to make any modifications[10]. This assumes

[6] https://www.man7.org/linux/man-pages/man7/namespaces.7.html

that both systems A and B share a common interface for interaction to the software component[10]. In the context of plugin systems for text editors, portability can be interpreted in one of two different ways:

1. Every individual plugin is seen as a software component. This plugin is portable, if it can be loaded into two instances of the same text editor running on different platforms.
2. The entire plugin system itself is seen as a modular software component of a text editor. It is portable if it can be integrated into different text editors and run across different platforms.

This work considers only the first scenario, in which portability refers to each individual plugin, because this scenario is less extensive and the portability of individual plugins is easier to measure. The following scores are used to measure plugin portability:

0 – Not portable The plugin is not portable between different platforms. It is theoretically and practically impossible to run the plugin on different platforms.

1 – Theoretically portable The plugin is theoretically portable between different platforms. In practice this might be very complex and costly, e.g. having to run each plugin in its own dedicated virtual machine.

2,3,4 – Portable with a runtime The plugin is portable between different platforms, but it requires a runtime on the target platform. Because these runtimes can vary from one plugin system to another, a score range from 2 to 4 is specified here. During evaluation the specific runtime has to be analyzed regarding its complexity and impact on the host system. A more lightweight runtime could also enable higher portability of the plugin system itself as described in Section 3.1.4.

5 – Portable by design The plugin is portable between different platforms without requiring a runtime on each host application. In practice this is very hard to achieve. Advanced technologies such as fat binaries, which are binaries that encapsulate compiled machine code for multiple different architectures, might be necessary.

Note that the most extreme scores 0 and 5 are very unlikely for any imaginable plugin system. 0 requires a plugin not to be portable at all, while 5 requires that a plugin is portable to different platforms and architectures which is near impossible to implement on a technical level.

3.1.5 Plugin language interoperability

Text editors and their plugin systems are highly individual software. Some users have personal preferences e.g. keybindings, macros or color schemes, while others may require tools such as language servers for semantic highlighting and navigation, or tools to compile and flash a piece of software onto an embedded device.

A lot of times plugin systems are used to overcome this challenge of high configurability. Major text editors and IDEs such as Neovim, VSCode, Emacs, IntelliJ or Eclipse provide plugin systems, some even in-built plugins. The advantage of implementing features as plugins is the reduced code complexity and size of the host application. Also providing a plugin systems with a publicly documented interface allows every user and developer to implement their own plugins for their own needs.

This property describes how interoperable plugins written in different languages are. In other words, the larger the set of languages is, in which a plugin can be written for a given plugin system technology, the better its plugin language interoperability is. The more languages are available, the less effort it requires for users to develop their own plugins without the need of learning a new (possibly domain-specific) programming language.

One could argue, that supporting a variety of different languages can result in higher interface complexity and less adaptability to new changes. Even though the complexity and adaptability of interfaces is another important property, which deserves its own rating, it will not be covered in this work.

0 – Domain-specific custom language Only a domain-specific language can be used to write plugins in. This language is specifically designed for given plugin system technology, which is why it is the least interoperable and might be the most unfamiliar and hardest for developers to learn and use.

1 – One language A single general purpose programming language is supported to write plugins in. There might be some developers who are already familiar with the language.

2 – Multiple languages A set of multiple languages is supported to write plugins in. The plugin system is able to abstract over multiple different plugin languages, so that the host application has only a single interface to communicate with plugins regardless of their specific language used.

3 – Build target for some languages The plugin system supports a compilation target for plugins, that is targeted by multiple compilers for multiple programming languages. For example the Java bytecode is a compilation target for the Java, Kotlin and Scala compilers.

4 – Build target for a variety of languages The plugin system supports a compilation target for plugins, that is targeted by a variety of different languages. This score differs from the previous score, that this score's compilation target is targeted by a considerably higher number of languages with more differences between them. Differences between languages could include dynamic vs. static typing, weak vs. strong typing, interpretation vs. just-in-time compilation vs. ahead-of-time compilation.

5 – Universal build target The plugin system supports a compilation target to which most software can be compiled directly, or which can be embedded indirectly by a compiled runtime. For example all source code is eventually compiled to native ISA instructions specific to some hardware and platform. Thus it is also theoretically possible to package source code such as Python or JavaScript source code and combine it with their specific runtimes inside a native plugin.

3.2 Choice of technologies & projects

An important step during a technology comparison is the correct selection of technologies. When choosing technologies, one has to be careful to not introduce any bias towards certain technologies. This work mainly focuses on plugin systems for text editors, so text editor plugin systems will be the majority of technologies. However it could also be interesting to compare text editor plugin system technologies to plugin system technologies for other applications.

To make an appropriate decision for text editor projects, the popularity of different text editors will be as a metric for selection. As a source for the most popular text editors and integrated development environments (IDEs), the StackOverflow Developer Survey 2024 will be used[11]. The platform StackOverflow is known as a forum for developers helping each other by asking questions and exchanging information regarding various technical topics such as programming languages, certain APIs, technologies, etc. Annually it organizes a survey open to all developers regardless of their background to gather statistics about used programming languages, salaries of developers, used development tools, etc. In 2024 a total amount of ~65.000 developers took part in this survey, which is why it can be considered fairly representative. The four most popular text editors and integrated development environments (IDEs) according to the survey are the following. They are described and also considered for evaluation in this work:

1. **Visual Studio Code** Visual Studio Code (abbreviated as VS Code) is a text editor designed for software development. It provides built-in basic features such as a terminal, version control or themes[7]. However it does not provide built-in integration for specific programming languages or technologies. In VS Code a lot of features are instead packaged as plugins (here: extensions) written in JavaScript which can be installed from a central marketplace. VS Code itself is also written in JavaScript. It is chosen for evaluation in this work, because it has successfully implemented a plugin system able to integrate a broad spectrum of plugins ranging from simple plugins for coloring brackets to highly complex plugins such as programming language integrations or plugins enabling interactive jupyter notebooks.
2. **Visual Studio (not evaluated here)** Visual Studio is an IDE, that is describes itself as "the most comprehensive IDE for .NET and C++ developers on Windows"[8]. It is not to be confused with Visual Studio Code, as Visual Studio provides more built-in features for developing, debugging, testing and collaboration between developers[9]. It also allows users to install plugins (here: extensions) written in C# from a central marketplace. However due to personal unfamiliarity with this IDE and the C# programming language together with the .NET ecosystem, this IDE will not be evaluated in this work.
3. **IntelliJ-family** Intellij IDEA is an editor made by Jetbrains for development of JVM-based languages such as Java or Kotlin[10]. According to the StackOverflow developer survey, it is ranked as the third-most popular IDE.

   IntelliJ's so called "Community Edition" is freely available and open-source, however Jetbrains also develops proprietary alternatives for other use cases. In fact most of these alternatives such as CLion for C/C++ development or WebStorm for web development are also based on the IntelliJ framework. There are also third-

[7] https://code.visualstudio.com/
[8] https://visualstudio.microsoft.com/
[9] https://visualstudio.microsoft.com/vs/features/
[10] https://www.jetbrains.com/idea/

party IDEs built upon the IntelliJ IDEA such as Android Studio for development of Android apps.

IntelliJ provides a plugin system where plugins can be written in Java or Kotlin, that can be used from all IDEs based on the IntelliJ framework. Thus the IntelliJ IDEA is not listed as a single IDE here, but rather a family of various IDEs all based on the same framework using the same plugin system technology.

4. Notepad++ Notepad++ is an open-source text editor focused around minimalism and efficient software[11]. The editor itself is written in C++ with exclusive support for the Windows operating system. It provides a plugin system for loading plugins in the form of dynamically linked libraries (DLLs)[12].

Besides these text editors and IDEs, this work also evaluates plugin system technologies for other application types. Multiple projects and technologies were considered, however due to their similarity to already chosen technologies, missing documentation and time-constraints for this work most of them were disregarded:

VST3 for music production During music production in a digital audio workstation, producers use plugins to simulate a variety of different audio effects, entire instruments or traditional analog devices such as compressors. One notable standard used for for these kinds of plugins is the open and free Virtual Studio Technology (VST) 3 standard developed by Steinberg[12]. It specifies a standard for technology on how to implement a plugin system technology between a digital audio workstation (DAW) as the host application and a plugin which applies some effects or produces an audio signal. For example a DAW could sends an audio stream to some plugin, then the plugin could apply an effect such as a simple equalizer on the audio signal and return the signal back to the DAW.

On the technical level VST3 plugins are DLLs on Windows, Mach-O Bundles on Mac and packages on Linux. The term package on Linux can have different meanings depending on the Linux distribution and package manager used.

The company behind the VST3 standard also provides a software development kit (SDK) for the C++ programming language and an API for the C programming language.

[11] https://notepad-plus-plus.org/
[12] https://npp-user-manual.org/docs/plugins/

Microsoft Flight Simulator (not evaluated here) Microsoft Flight Simulator is a simulator for aircraft with focus on ultra-realism[13]. It allows for a wide variety of aircraft, airports and systems such as advanced flight information panels. This is achieved by providing multiple software development kits (SDKs) for plugins (so called add-ons). These SDKs support plugin languages such as C, C++, .NET languages, JavaScript or WebAssembly[14].

However this software project will not be evaluated in this work. It's source code is not publicly available, which makes analysis of its architecture harder. Also a paid license is required to make an appropriate evaluation possible.

Eclipse(not evaluated here) Eclipse is an IDE used during software development for a variety of programming languages. It features a plugin system where plugins are written in Java and a central marketplace for installing plugins. However Eclipse is not chosen for evaluation, because its plugin system is too similar to IntelliJ's. Also IntelliJ is more popular as an IDE [11] and thus required to support a greater variety of plugins.

3.3 Evaluations of technologies & projects

3.3.1 Visual Studio Code

Performance In terms of performance Visual Studio Code performs reasonably well. It builds on web technologies, specifically Electron which utilizes the Chromium browser engine together with Node.js. VS Code is written in JavaScript, which still imposes some limitations compared to native applications such as pauses due to garbage collection or the single-threaded nature of Node.js modules[15]. While JavaScript engines such as the V8 engine try to mitigate most issues, the performance of VS Code still remains slower and more memory-hungry than native alternative IDE and text editor applications written in C, C++, Rust, etc.

Even though the overhead imposed could be considered small and VS Code and it's plugin system is quite fast for today's standards, its performance is still rated with a score of 3. This score is chosen, because VS Code performs below average but still acceptable in comparison to typical native applications.

[13]https://www.flightsimulator.com/
[14]https://docs.flightsimulator.com/html/Programming_Tools/Programming_APIs.htm
[15]https://www.electronjs.org/docs/latest/tutorial/multithreading#native-nodejs-modules

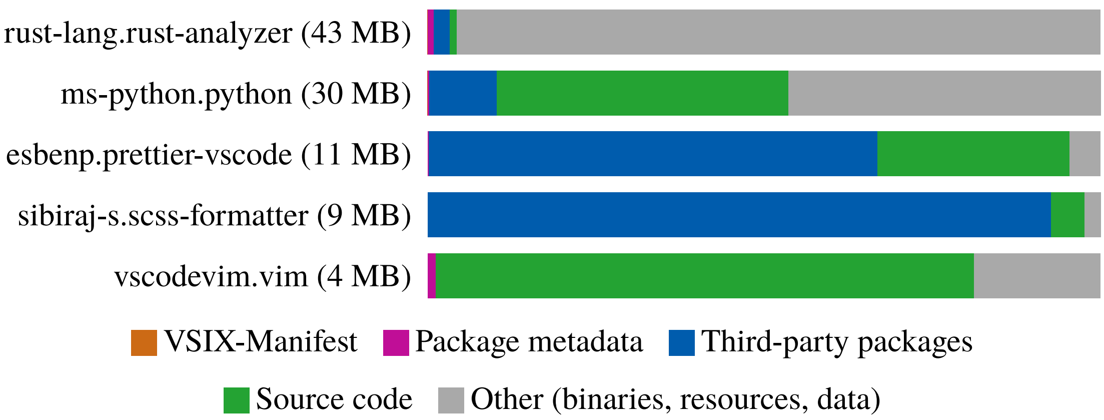

Figure 4: Plugin size distributions of selected VS Code plugins

Plugin size The average plugin size for VS Code plugins is highly variable. Most minimal plugins typically consist of JavaScript or TypeScript code and a JSON manifest file. However, more complex plugins can be significantly langer, for example plugins that bundle third-party libraries and dependencies or language servers.

Because VS Code is based on web technologies, its plugins can also be used in web environments, where its dependencies have to be bundled, which can further increase plugin sizes. Though on average, the size of VS code plugins a lot due to the size of libraries and resources needed for specific use cases.

Figure 4 shows plugin sizes of various arbitrarily selected VS Code plugins. The largest plugin here is the rust-analyzer plugin with a size of 43MB. The smallest is the Vim emulation plugin with 4MB. The diagram also shows the distribution of different data types and their contribution to each plugin's total size. Here one can see that some plugins contain mostly other data, such as the rust-analyzer plugin which contains the binary rust language server. While other plugins make use of third-party packages such as external dependencies or libraries, such as the SCSS Formatter plugin, there are also plugins with a high share of source code such as the Vim emulation plugin. However for the evaluation of the average plugin size for the VS Code plugin system, other data such as resources, images and binaries must be left out, because this data varies from one plugin to another. Looking at the diagram, one can see, that the size of source code around the high KBs to the low MBs. Thus this plugin system technology is evaluated at a score of 3.

Plugin isolation

VS Code provides a moderate level of plugin isolation. It is based on web technologies such as JavaScript. A lot of web technologies like JavaScript are designed to enable safe code execution on a client's browser. This is achieved through sandboxing, which means running an application in an isolated "sandbox", where the application cannot escape this sandbox.

VS Code runs all plugins (officially known as extensions) in separate extension host processes to isolate them from the main application[13, sec. Extension runtime security]. Also plugins do not gain direct access to the main editor's state such as the DOM tree. Instead only specific APIs are exposed to the extension host.

However the extension host inherits the same permissions as the original VS Code application. This means that plugins running together in a process with the extension host may have full access to filesystems, peripherals or running processes. Because of this inheritance of permissions and the dynamic and flexible nature of JavaScript itself, malicious plugins are not necessarily restricted from accessing the main editor state.

Thus VS Code's plugin isolation is evaluated at a score of 3. Plugins run isolated from the main application, however malicious plugins may be able to circumvent this isolation through the inherited host privileges.

Plugin portability VS Code plugins are very portable. Since they are build on web technologies, that are required to be portable, plugins can run on various platforms, operating systems and architectures mostly without modifications.

That said, plugins that rely on native binaries such as language servers, which are often compiled in advance and then shipped with the plugin, may require separate build for different platforms, reducing portability.

In summary the portability of VS Code plugins is still very high, except for some edge cases, relying on native code, resulting in a plugin portability score of 4.

Plugin language interoperability Plugin language interoperability describes how many programming languages may be used to develop plugins. VS Code only allows JavaScript or TypeScript, which builds on JavaScript with a type system, as plugin languages.

- asm.js is subset of JS: acts as a compilation target for C/C++ with Emscripten
- Also plugins may include native code. This allows them to ship compiled programs written in C, C++, Rust or entire compiled runtimes for languages such as Python.

In such scenarios a plugin's JavaScript code could contains only glue code to forward function calls between the plugin core written in C, C++, Python, etc. and the VS Code host application.

While it is possible to write parts of VS Code extensions in other languages than JavaScript or TypeScript, the complexity of such plugins may increase rapidly. However following a strict evaluation for VS Code's plugin system, it receives a score of 1, because the plugin system always requires plugins to contain some JavaScript code, even if it is just glue code.

3.3.2 IntelliJ-family

Performance IntellIj IDEs also perform reasonably well. Both the IDE, as well as its plugin system and plugins are written in languages targeting the Java Virtual Machine (JVM). That is, languages such as Java, Kotlin or Scala, which are optimized and compiled to Java bytecode by their respective compilers. This Java bytecode can then be executed by the JVM. During execution the JVM can apply additional optimizations and generate native machine code for execution for example through optimizing Just-in-time (JIT) compilation, only then when it is needed.

Even though JVMs are highly complex systems with loads of mechanisms for optimization, they still introduce some overhead. Also they are responsible for holding the state of programs during runtime and for garbage collecting no longer used object instances.

Thus the performance of plugin systems and plugins based on the JVM, will not be able to outperform native non garbage-collected approaches. This plugin system technology, as it is used in IntelliJ-based IDEs is rated with an average performance score of 3.

Plugin size Plugin sizes for IntelliJ plugins also vary a lot. IntelliJ plugins are Java Archive (JAR) files, that can contain source code in the form of class files, resources or any arbitrary data. Small plugin JARs might contain only a few source files, while larger plugins might rely on various dependencies, which are all compiled and statically linked into the plugin JAR file.

For IntelliJ plugins an extensive analysis of plugin size distributions was not possible due to the complexity of internal plugin structures. However some plugins were selected as examples for their total size: The official plugin for Rust language support has a size of 54MB, the official Python language support plugin 69MB and the Prettier formatter plugin has 508KB. Thus the average plugin size for the IntelliJ plugin system is evaluated at a score of 3.

Plugin isolation Both the IntelliJ IDEs and plugins are written in JVM-based languages. No documentation or information was found on the security of IntelliJ plugins. Thus is it assumed, that IntelliJ plugins are loaded into the same JVM as the host IDE, inheriting the host's privileges.

Because there are no isolation mechanisms in place, the plugin isolation of plugins for IntelliJ-based IDEs are evaluated at a score of 1.

Plugin portability Since IntelliJ plugins are written in Java or other JVM-based languages, they benefit from Java's cross-platform portability. This allows plugins to run consistently across different environments. Because IntelliJ itself is build on the same technology, this enables every plugin to run on every platform, as long as plugins do not introduce platform-specific behavior such as relying on native binaries. Plugin portability is rated at a score of 4 for all IntelliJ-based IDEs, as Java bytecode is portable because the JVM is still required as a runtime on the target platform.

Plugin language interoperability IntelliJ supports all plugins that can be compiled to Java bytecode, as long as they adhere to the IntelliJ Platform API, which can vary from one IntelliJ-based IDE to another.

While only Java and Kotlin are officially supported, other JVM-based languages such as Groovy or Scala could also be used, as they also compile to Java bytecode. This plugin system, which supports one compilation target used by a variety of languages, is evaluated at a score of 4.

Figure 5: Plugin size distributions of selected Notepad++ plugins. They were analyzed by running `winedump` on a Linux system on the plugin DLL files. Then the `size of code` and `size of initialized data` fields were extracted to show here.

3.3.3 Notepad++

Performance Notepad++ is written in C++ and compiled to native machine code for the Windows operating system exclusively. It tries to maximize efficiency and minimize the impact on the system it is running on. Its plugin system is based on compiled dynamically linked libraries (DLLs). A plugin developer compiles their plugin, which can be written in any arbitrary language, to a DLL, which is a library containing machine code and lists of imported and exported symbols. Notepad's plugin system can then load theses libraries at runtime via the `LoadLibrary` Win32-API call and execute arbitrary functions exported from the plugin.

This is the fastest way for how host applications can embed plugins. It relies only on the operating system for loading already compiled machine code at runtime. There is no additional overhead except having to load the DLL itself into memory. Thus Notepad++'s plugin system is evaluated at a score of 5 for its optimal performance.

Plugin size Notepad++ uses DLLs as its plugin format, meaning plugins are compiled directly to machine code. These binaries may include statically linked libraries and dependencies, though some exceptions for example for `libc` might exist. Figure 5 shows selected plugins and their total sizes and size distributions. The size distribution into machine code, resources and other data was extracted from the respective DLLs through use of `winedump` on a Linux system. The bar chart shows, that most data inside each plugin comes from machine code and not from any other resources or binaries. Given this analysis and plugin sizes being under

500KB for all plugins, a rating of 4 is assigned to this plugin system technology. This high rating is due to the compactness of machine code, which generally is generally minimized by compilers for better runtime performance already. In practice smaller binaries require less data transfer rates between the memory and the CPU and thus result in higher cache hit rates and better performance.

Plugin isolation However Notepad++‘s plugin isolation is rated poorly at a score of 1. Since Notepad++ loads plugins as dynamic libraries at runtime, these libraries inherit all privileges from the host editor. There is also no sandboxing in place, so plugins essentially run on the exact same level as the host editor. This is a severe risk, because the user has to trust both the plugin developer and also has to verify the integrity of a downloaded plugin.

Plugin portability Plugin portability for Notepad++ plugins is rated at a score of 1. Notepad++ itself supports only Windows 8.1 - 11 as an operating system for multiple architectures[16]. Plugins have to be compiled and shipped for every supported architecture separately. Thus plugins are compatible in theory, because one could run plugins compiled for another architecture inside a virtual machine packaged as a plugin itself. However this is not feasible in practice due to performance losses and the complexity involved.

Plugin language interoperability With native machine code as plugins, Notepad++ allows for great plugin language interoperability. In practice it common to write plugins in languages that compile directly to machine code, such as C, C++ or C#. Notepad++ provides official header files for the API to Notepad++ for plugin development in C++. There are also community-driven API definitions and templates for plugin development in Ada, C#, D, Delphi or Ada available[17].

One can also argue, that every piece of software running today is eventually compiled to or interpreted as machine code. Thus it may be possible to embed higher-level technologies such as a Python runtime along with plugin logic written in Python inside a plugin.

Because native machine code is a universal build target for all languages and technologies and the resulting versatility, this plugin system is rated at a score of 5 in terms of plugin language interoperability.

[16] https://github.com/notepad-plus-plus/notepad-plus-plus/blob/master/SUPPORTED_SYSTEM.md

[17] https://npp-user-manual.org/docs/plugins/

3.3.4 VST3

Performance The VST3 standard for plugins that create or process audio relies on native machine code just like Notepad++. However it's file format also accommodates for the fact that plugins might be run on more than one platform. Internally a VST3 file embeds either a DLL for Windows plugins, a Mach-O bundle for MacOS plugins or generic packages for Linux plugins.

During runtime a host application has to check whether a VST3 plugin contains machine code compiled for the current architecture. Then it is able to link the machine code at runtime through the operating system just like Notepad++ does.

While there is a small overhead of checking if the targeted platform of a plugin is correct for loading a plugin, there is no overhead during execution of plugin code. Thus the VST3 technology is evaluated at a score of 5.

Plugin size Because VST3 plugins are distributed as platform-specific binaries, their size can also remain relatively small comparable to Notepad++'s plugins. On the other hand, VST3 plugins often contain more resources such as images for providing the user with a graphical user interface.

Depending on the specific use case, one could rate VST3 plugin sizes at a score of 3 or 4. However to enable objective comparison and VST3's strong similarity to Notepad++, plugin size is rated at a score of 4.

Plugin isolation VST3 plugins are loaded as dynamic libraries just into the host's memory space inheriting privileges comparable to Notepad++'s plugin system. There is no isolation between the host application and VST3 plugins, which can be used by malicious plugins posing a significant risk to the end-user. Thus plugin isolation is rated at a score of 1.

Plugin portability VST3 plugins itself are not portable. They are compiled for every architecture and operating system and then distributed separately. Even though the VST3 documentation states that the source code of plugins can be reused across multiple platforms, the plugin portability stays unaffected as this benefits just the plugin developer. The portability of VST3 plugins is therefore rated at a score of 1.

Plugin language interoperability VST3's plugin language interoperability is also similar to that of the Notepad++ plugin system technology. It's plugins are also based on native machine code, and thus it is possible in theory to compile or embed most technology into a plugin. However in practice it is more common for

	Performance	Plugin size	Plugin isolation	Plugin portability	Plugin language interoperability
Visual Studio Code	3	3	3	4	1
IntelliJ-family	3	3	1	4	3
Notepad++	5	4	1	1	5
VST3	5	4	1	1	5

Table 1: Technology comparison matrix for selected technologies and software projects

plugin developers to develop plugins with C++ using the official VST3 software development kit. The universality of the native binary format earns VST3 a rating of 5 for plugin language interoperability.

3.4 Results & summary

Three different plugin systems and one standard for a plugin system technology have been evaluated with the previously defined criteria. The results can be seen in a technology comparison matrix in Table 1. Here every technology row is evaluated across all five criteria, namely performance, plugin size, plugin isolation, plugin portability and plugin language interoperability. Each criterion is rated on a scale from 0 (very poor) to 5 (excellent), however the smallest score that can be found here is 1.

Visual Studio Code's plugin system performs with average scores of 3 across performance, plugin size and plugin isolation. It provides good plugin portability from its usage of web technology, but is not able to provide a lot of plugin language interoperability due to the limitation to JavaScript/TypeScript. The plugin system used in all **IntelliJ-based IDEs** achieves similar scores for performance and plugin size, however it does not provide any plugin isolation. It is able to achieve the same plugin portability as VS Code and a higher score for plugin language interoperability because it allows any JVM-based languages for plugins. **Nodepad++** is another text editor with efficiency in mind. This shows, because it achieves the highest score for performance with plugin sizes that are negligible in practice. However it provides terrible plugin isolation and plugin portability due to plugins consisting of natively compiled machine code. On the other hand, native machine code also acts as a universal compilation target, allowing pretty much every other technology to be compiled into a plugin. This allows for an optimal plugin language interoperability. **VST3** is a standard for a plugin system technology. It follows the same approach as Notepad++'s plugin system with native machine code inside its plugins. Thus it achieves the same scores, although in practice the VST3 format is merely a wrapper around native machine code, providing additional features not considered here.

Now a critical discussion of the chosen criteria is presented, outlining possible improvements and other relevant criteria, not covered in this section. Some criteria, namely the average plugin size and performance, have been analyzed only qualitatively due to time constraints. However, these criteria can and should be analyzed quantitatively to provide more accurate and verifiable results. There are also criteria that may also be important for plugin systems in general, not covered here:

Complexity and flexibility of plugin system interface Plugin systems with more flexible and dynamic interfaces without a fixed interface definition may enable faster development and testing of new features.

Developer experience Developer experience could be used as a measure for how easy it is for developers to develop plugins.

Debuggability Some technologies might be easier do debug and inspect during runtime. This allows plugin developers to spend less time searching for bugs, which then results in a faster plugin development.

Plugin system size This section analyzed the average size of plugins. The size of entire plugin systems was not considered. However when plugins get smaller, the size of plugin systems could increase, due to logic being outsourced from plugins to the plugin system. Thus to optimally evaluate sizes, one should consider both the plugin size and plugin system size and not just rely on one.

To conclude, four plugin system technologies were compared and the results visualized in a technology comparison matrix. While the matrix shows the strengths and weaknesses of these technologies across different criteria, there is more to consider when choosing an appropriate plugin system technology. In the future, additional criteria should be defined for a more accurate comparison.

4 WebAssembly for plugin systems

TODO: Present basic idea of running Wasm code for each plugin inside a Wasm runtime

4.1 Overview of basic plugin system architecture

4.2 Evaluation of requirements

4.3 Evaluation of interface-specific requirements

It is not possible to evaluate Wasm as a technology, because it only provides fundamental technology for executing WebAssembly program code. In practice WebAssembly is often combined with other technologies that build upon Wasm's basic constructs and allow for more complex systems such as type systems or interface definitions.

WebAssembly as a technology is often not restrictive enough and thus the decision of whether a requirement is fulfilled often comes down to the host- and plugin-language and whether a common interface definition between them exists.

To illustrate this point, consider a scenario in which a host system is written in JavaScript. When this host system wants to call a Wasm function it serializes the arguments, which might consist of complex JavaScript types, to JSON strings. Then it passes these JSON strings to the Wasm plugin. A plugin written in JavaScript itself will be able to easily parse the JSON string given to it, however a plugin written in C first has to get a system in place to parse and convert JavaScript types to equivalent C types.

TODO: Wasm interfacing is still an unsolved problem. There are many different solutions, of which some will be evaluated separately here

4.3.1 Without a standardized interface

TODO

4.3.2 WebAssembly Component Model

TODO

4.3.3 WebAssembly Component Model + WebAssembly System Interface (WASI)

TODO

4.3.4 Custom serialization format (JSON, XML, Protobuf)

TODO

4.4 Plugin systems already using WebAssembly today

Zed text editor with Wasm plugin system, no windows support

- Wasm, but official interface only for Rust plugins?
- WASM mit kompletter WIT Schnittstellendefinition (Generisches Typ/Funktions-basiertes System)pro Version
- Fokus auf Isolation von WASM Code
- shim library nur für Rust vorhanden: zed_extension_api

https://zed.dev/docs/extensions/developing-extensions https://github.com/zed-industries/zed/blob/94faf9dd56c494d369513e885fe1e08a95256bd3/crates/extension_api/wit/since_v0.2.0/http-client.wit

Zellij terminal multiplexer, has a Wasm plugin system.

- Wasm, but official interface only for Rust plugins?

Zellij is a terminal workspace (similar to a terminal multiplexer). It is used to manage and organize many different terminal instances inside one terminal emulator process. Similar commonly known terminal multiplexers are Tmux, xterm or the Windows Terminal.

- plugin system to allow users to add new features
- plugin system is not very mature https://zellij.dev/documentation/plugin-system-status
- Wasm for plugins, however only Rust is supported
- Permission system

Extism generic Wasm plugin system library usable in many different languages

- Extism is a cross-language framework for embedding WebAssembly code into a project.
- It is originally written in Rust and provides bindings (Host SDKs) and shims (Plugin Development Kits: PDKs) for many different languages.
- Docs: https://extism.org/docs/overview

- Github: https://github.com/extism/extism

go-plugin *TODO*

4.5 Summarized evaluation for WebAssembly

TODO: Show all WebAssembly configuration in a matrix with all criteria as columns

5 Proof of concept: Implementing a WebAssembly plugin system for a text editor

TODO: Provide context of helix text editor

5.1 Requirements

TODO: Weight the requirements for plugin systems and optionally add new requirements for this project

5.1.1 Functional requirements

5.1.2 Non-functional requirements

5.2 System Architecture

TODO: Choose appropriate Wasm technologies based on the previous findings and document how they are used together to build a working system

5.3 Implementation

TODO: Give brief overview over code structure TODO: Technical challenges and how they were addressed TODO: What optimizations were made?

5.4 Evaluation & Results

TODO: Reevaluate the key requirements for plugin systems TODO: Provide measurements for memory/performance impact (there are no real reference points) TODO: Summarize findings & challenges

Standard plugin definieren (z.b. textsuche für performance) Graph mit x geladenen Standard plugins für memory/performance

6 Results & Discussion (2 pages)

- some of wasms strenghts were not represented during the technology comparison. namely native-like safe interfacing

7 Outlook

Bibliography

[1] A. S. Tanenbaum and T. Austin, *Structured Computer Organization*, 6th ed. Pearson Education, Inc., 2025.

[2] A. Haas *et al.*, "Bringing the Web up to Speed with WebAssembly," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2017*, Association for Computing Machinery, Jun. 2017, pp. 185–200. doi: 10.1145/3062341.3062363.

[3] W. Zaeske, S. Friedrich, T. Schubert, and U. Durak, "WebAssembly in Avionics: Decoupling Software from Hardware," in *IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, Nov. 2023. doi: 10.1109/dasc58513.2023.10311207.

[4] M. N. Hoque and K. A. Harras, "WebAssembly for Edge Computing: Potential and Challenges," in *IEEE Communications Standards Magazine*, IEEE, 2022, pp. 68–73. doi: 10.1109/MCOMSTD.0001.2000068.

[5] S. Wallentowitz, B. Kersting, and D. M. Dumitriu, "Potential of WebAssembly for Embedded Systems," in *11th Mediterranean Conference on Embedded Computing (MECO)*, IEEE, Jun. 2022. doi: 10.1109/meco55406.2022.9797106.

[6] WebAssembly Community Group and A. Rossberg, "WebAssembly Core Specification (Release 2.0, Draft 2025-01-28)." Accessed: Mar. 27, 2025. [Online]. Available: https://webassembly.github.io/spec/core

[7] "bla."

[8] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-Safe Multilingual Software Sandboxing using WebAssembly," in *31st USENIX Security Symposium (USENIX Security 22)*, USENIX Association, Aug. 2022. Accessed: Apr. 15, 2025. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya

[9] B. Spies and M. Mock, "An Evaluation of WebAssembly in Non-Web Environments," in *2021 XLVII Latin American Computing Conference (CLEI)*, 2021. doi: 10.1109/CLEI53233.2021.9640153.

[10] M. van Steen and A. S. Tanenbaum, *Distributed Systems: Principles and Paradigms*, Edition 4, version 03. Maarten van Steen, 2025. Accessed: Mar. 27, 2025. [Online]. Available: https://www.distributed-systems.net/index.php/books/ds4/

[11] "StackOverflow Developer Survey 2024." Accessed: Apr. 08, 2025. [Online]. Available: https://survey.stackoverflow.co/2024/

[12] “VST 3 Developer Portal.” Accessed: Apr. 09, 2025. [Online]. Available: https://steinbergmedia.github.io/vst3_dev_portal

[13] “Visual Studio Code documentation.” Accessed: Apr. 13, 2025. [Online]. Available: https://code.visualstudio.com/docs