Electron and *Tauri*: Comparison of web-frameworks for building cross-platform desktop applications

Kevin Lüttge
University of Kassel

ABSTRACT

Web-frameworks like Electron provide the possibility to easily develop cross-platform desktop applications by using common web-technologies such as Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript. The increasing popularity of those frameworks is shown by various day-to-day applications namely WhatsApp, Discord or Visual Studio Code that are built with Electron. Two years ago Tauri came up, whose developers claimed it to be faster, smaller and more focused on security than other alternative frameworks. This paper will discuss the differences and similarities between Electron and Tauri, in terms like architecture, performance or security. Therefor a counter application is implemented for each framework to analyze and compare the development process.

1. INTRODUCTION

Cross-platform frameworks like Electron, Tauri, Flutter or Proton are providing the possibility of using web-technologies like HTML, CSS and JavaScript or whole web-frameworks like Angular or React to develop or migrate classic desktop applications to web applications. The classic approach of implementing desktop applications needs the developer to consider different Application Programming Interface (API) or environments of the major operating systems Operating System (OS) Windows, Mac and Linux. In fact each application has to be implemented multiple times to adapt OS-specific requirements. In context of consumer applications this allows companies and product owners to develop new target groups, but also increases the pool of potential applicants, since those frameworks provide web developers to implement desktop applications without the knowledge of standard programming languages that are used for desktop applications like C/C++ or Java. But also institutions or universities and their students benefit of it, because due to the fact of cross-platform development, applications can be used by many devices and decrease costs since they do not need particular hardware.

1.1 Background

Over the past years frameworks for building desktop applications have get more and more attention. This is owed to the fact of fundamental advances at web technologies like TypeScript or Frontend Frameworks like Angular or React [9], which lead to a wider use of such technologies for various scenarios. A lot of every day applications used by computer scientists like Visual Studio Code or GitHub Desktop, but also applications with usage spread over industry sectors like Microsoft Teams, Skype or Discord and even Social Media applications like WhatsApp or Twitch are implemented with such frameworks. The trend of using those applications has even grow up since the outbreak of SARS-CoV-2 in early 2020 [5] and the increased number of employees working from home as a result of lockdowns all over the world. This lead to an increasing number of cross-platform web-frameworks with different approaches in context of building, security or performance.

1.2 Motivation

As mentioned in Chapter 1.1 various cross-platform web-frameworks came up using different technologies or languages. One of the oldest and mostly used frameworks is Electron, which is backed up by several popular applications implemented with Electron like Visual Studio Code, Discord or Twitch. Therefore, Electron has become a standard in context of cross-platform development of desktop applications and almost every new framework invented is benchmarked against it like [7] or [3]. Two years ago a new framework called Tauri was introduced which is designed to improve several aspects of Electron especially in case of performance, memory usage and security [1]. Since these statements are made by the developers this paper aims to provide an objective overview to the reader of both frameworks Electron and Tauri. Therefor fundamental architecture as well as the frontend and backend core of each framework is explained in detail at chapter 2 for Electron and at chapter 3 for Tauri. This technological knowledge is applied by implementing a basic application which will be described in detail at chapter 4 and analyzes different aspects. To obtain an objective statement of the advantages and disadvantages of each framework chapter 5 compares the results of previous sections to isolate advantages and disadvantages to the reader. At the end of this paper the results of previous chapters are contrasted against the statement of the Tauri Developers in [1] and being discussed to obtain an objective comparison.

1.3 Related Work

An exploratory study of [12] has worked out, that applications developed with cross-platform web-frameworks are made for various kinds of usage. The authors also empirically documented that those frameworks are mostly used by developer teams with a median size of 1, which is a direct result of the amount resources classical desktop development consumes. Nevertheless, they also discovered disadvantages like a high number of used libraries due to the fact of compensating lack of features provided by web-frameworks, but also a high ratio of platform-related issues to all issues of 20%

The increasing popularity of using web-technologies has been shown by [9]. Since they identified their native Java Swing desktop application as a bottleneck, the authors decided to replace it with a technology stack providing long term sustainability. Therefor they used several web-technologies like AngularJS or Typescript to implement "a web-based tool for configuring experiments on the National Ignition Factory". But they also came in touch with typical problems of the JavaScript ecosystem like exit or replacement of widely used technologies bringing them to be forced to migrate AngularJS to Angular. It has to be mentioned that the authors emphasized the community support especially in case of migration by providing tools those cases.

Electron as a standard benchmark for cross-platform web-frameworks is shown by [3] or [7]. Both authors compared different web-frameworks against Electron. Although the benchmark of the authors of [3] was made with different Integrated Development Environment (IDE)s and the comparison between Flutter and Electron by [7] was based on beta version of Flutter both found out that at the one hand Electron has better performance in case of execution time and CPU consumption and on the other that it provides more features than the compared frameworks.

2. ELECTRON

Electron originally was released as Atom Shell by GitHub at 15$^{\text{th}}$ July, 2013 [17] and downloaded approx. 83 000 000 times[1] The intention of the developers was "to handle the Chromium/Node.js event loop integration and native APIs" [11] for the Atom Editor. On 23$^{\text{rd}}$ April, 2015 it was renamed to Electron and announced that the developers want to provide a framework that allows to build desktop applications with the using web-technologies. Many day-to-day apps have been implemented using Electron like Discord, Twitch or even Microsoft Teams. This has lead to an increasing community which can be expressed numerically based on GitHub Statistics [14]:

Stars	Forks	Watching	Used by	Contributors
130 000	13 700	2 900	244 568	1 126

It combines both Chromium, an open-source browser and Node.js, an open-source JavaScript-runtime, to provide frontend-based features like rendering HTML as well as os-based features like access to the filesystem inside one framework.

[1]According to `https://npm-stat.com/charts.html?package=electron&from=2013-07-13&to=2022-07-30`

2.1 Architecture

Electrons architecture fundamentally relies on Chromium, which is included in each Electron executable and has a lot common with it. As Chromium, Electron is based on a multiprocess model, containing of a single process called **main** and several processes, one for each window, called **renderer**. Figure 1 shows such a multiprocess model structure, where the `main` process creates and manages multiple `renderer` processes [6, Chapter 1.5].

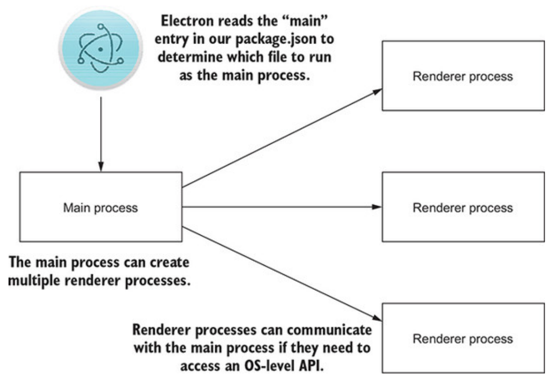

Figure 1: Multi-process Model Electron from [6, Fig. 1.7]

Main:
This is the main entry point of the application and responsible for lifecycle management like starting or quitting the app. The `main` process uses so called `BrowserWindow` module to create and manage each `renderer` process which is loading web pages into it. Since just the `main` process is running inside a Node.js environment, it is the only part of the application that can import Node.js modules using `require`. This forces each `renderer` process to interact with the `main` process if they want to consume system APIs for purposes like saving files or opening dialogs [15][6].

Renderer:
A `renderer` process is responsible for rendering web content by loading web pages into it and presenting them to the user. Additionally, JavaScript code can be loaded and executed inside a process. Each `renderer` process can be created or destroyed by the main process using the `BrowserWindow` module as mentioned before. This leads to the fact, that `renderer` processes are isolated from each other following the Chromium principles of a multiprocess model and is reasoned by limited affection of faulty or malicious code on the entire app. The `renderer` processes are only able to communicate between each other indirectly via the `main` process or by using `MessagePorts`. This is called Inter-Process Communication (IPC) [15][6].

2.1.1 Inter Process Communication

As mentioned above `main` and `renderer` processes are only able to communicate using IPC. Therefor Electron provides two modules, one for the `main` process, called `ipcMain` and one for `renderer` process, called `ipcRender`. Both of them are Node.JS `eventEmitter` modules and capable of executing asynchronously and synchronously communication either uni- or bidirectional. It should be noticed, that this kind of IPC communication is only for `renderer` to `main` and vice versa. A communication between different `renderer` processes can be archived either indirectly using the `main` process or directly with `MessagePorts`. The advantage of using the direct variant is, that the workload of the `main` process can be reduced by using a new, hidden `renderer` process, which serves as a worker for the web content `renderer`. A `MessagePort` can be seen as a communication channel between two `renderer` processes. It is once defined inside the `main` process which informs the `renderer` processes about belonging `MessagePorts` from other processes and establishes the connection. After that, the actual communication does not relay at the `main` process anymore. This will reduce workload for the `main` process in case of message forwarding as well as outsourcing heavy workloads to worker processes [15].

2.1.2 Context Isolation

Electrons multiprocess model also intends distinct purposes for each process. As described before only the `main` process has access to node modules. In contrast to that only the `renderer` process has access to HTML Document Object Module (DOM)s. Since using just IPC represents a major security issue, Electron introduced a feature called **Context Isolation**. This splits the logic of a `renderer` process into two different contexts. On the one hand the `renderer` process as already described and on the other a so called `preload script`. The `preload script` is attached to the `main` process at the creation of the `BrowserWindow` module. It has access to both node modules and HTML DOMs at its own context and will be executed before the `renderer` process loads the web page into it. Although the `preload script` and the `renderer` process do both own a window object, which provides the displayed browser window, it is not the same object since they are both running at different contexts. The `preload script` consists of a `contextBridge` module which is responsible for safely exposing selected properties of the `main` process to the `renderer` and vice versa. Inside this module an API can be defined for providing access with IPC objects to resources of different processes [15].

2.2 Frontend

The frontend part of an electron-based app consists of the chromium content module and provides all the core features that are required to render HTML content or access Web-APIs. It is important to notice, that the content module is not equal to the chrome web-browser since the chrome web-browser wraps around the content module, but provides several features, the content module does not provide, like managing bookmarks, spellchecking, safe-browsing or securely saving passwords. The content module itself includes two parts. A browser engine called **Blink** and a JavaScript engine called **V8** [6].

Blink:

Blink serves as a browser engine inside the chrome content module and therefor is responsible for translating HTML documents to the actual view, a user gets presented. Originally chromium used webkit as browser engine, which was developed and maintained by Apple, but fundamental disagreements between Apple and Google and Apple's restricted policy led to the fact, that Google forked the webkit engine and use it as a grounding for their own engine [4][10]. It relies on the same architecture as mentioned in 2.1, whereas blink is running inside a `renderer` process and uses one main thread and multiple worker threads that do the layout calculations.

V8:

V8 is written in C++ and responsible for compiling and executing JavaScript code as well as memory management like allocation or garbage collecting. It can be seen as the executing background part of the content module, which transforms JavaScript code into C++ code, which in turn is translated into machine-readable byte-code, but it also provides the possibilities for developers to write own C++ applications that can expose their functions to JavaScript Code to add new features or improve performance of execution [2]

Summarized the content module provides all features to Electron that are necessary to develop a classical browser application. But since classical browser applications are restricted by the OS for example in case of file-system access, the use-cases are limited. This is the reason, why Electron has combined the content module with the Node.js runtime.

2.2.1 Backend

As backend for all electron-based applications the Node.js runtime is used. Node.js is a framework allowing developers to implement server-side applications using JavaScript, and as the chromium content module of 2.2 it uses the V8 JavaScript engine to execute JavaScript code, but also provides more functionalities like the mentioned filesystem access or importing external modules. Furthermore, a package manager, called Node Package Manager (NPM), with "more than one million packages"[2] is included. This amount of packages is a result of an increasing popularity Node.js experienced since its release, because it can be used to implement a wide-range of applications. The reason for using Node.js runtime as a backend for all Electron applications is its privileges. At classical web-applications, the client-side is restricted through the OS to consume or request data from a third party API. Because of that, the client has to request the server, which then is consuming the third party API. Since Node.js has these privileges, the detour at the server-side falls away, guaranteeing electron applications these accesses at the client side.

3. TAURI

Tauri was first released at 31^{st} December, 2019 [8] and downloaded approx. 58 000 times[3]. It resulted as discontent of Electron from the Tauri Developers especially in case of resource consumption and security due to uncontrollable dependencies. They criticized the enormous resource consumption even of the simplest applications as well as the

[2] `https://www.npmjs.com/`

[3] According to `https://npm-stat.com/charts.html?package=tauri&from=2019-12-31&to=2022-08-12`

fact, that Electron does not have control over their dependencies. If Chromium encounters a zero-day-exploit and releases a patch, Electron has to include this patch and also release a newer versions. This leads to the fact, that users of an Electron-based application have to update their whole electron version to close this security issue. This timespan, between first fix of chromium and the users update of the applications, is a high vulnerability for attackers. In this regard, the developers also criticized the power of the privileges Electron applications have, allowing attackers to have access to the entire hard drive of the user for example [1]. But like Electron Tauri experienced increased attention, which as in 2 can be expressed numerically based on GitHub Statistics [13]:

Stars	Forks	Watching	Used by	Contributors
48 200	1 200	403	3 200	182

But unlike Electron, Tauri uses the self developed core, called Tauri, which is written in Rust in combination with WRY, which serves as the rendering library.

3.1 Architecture

The architecture of Tauri is very similar to Electrons multiprocess model. Tauri also relies on a main process, called **core** process and multiple rendering processes, called **webview** for performance as well as security reasons. Figure 2 shows the basic architecture of Tauris multiprocess model whereas each `webview` process is managed by the `core` process. by the

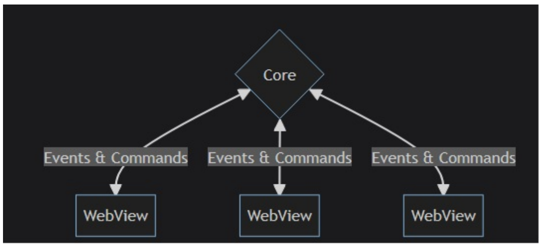

Figure 2: Multi-process Model Tauri from [1]

Core:
This is the main entry point of the application and the only place where full access to the operating system is provided. The `core` process uses this privileges to create and manage the windows the application. But it is also the only point, where the communication between processes is going through, allowing the core process to manipulate or observe IPC messages. Additional the `core` process is responsible for global scoped cases such as database access or managing application or os-specific settings that affect the windows. Summarized it serves as a centralized management and control point, where the application as itself is maintained and sensitive data is kept to be hidden from the `webview` processes.

WebView:
A `webview` process renders the ui by using the WebView libraries of the current os. Since this library is not included into the final executable but linked at runtime, it differs depending on the operating system the application is executed. This reduces the size of the executable since the part where the actual rendering takes places is shifted from the application to the operating system but also results that developers have to keep in mind the different operating systems.

3.1.1 Inter Process Communication

For communication between different processes Tauri also uses Inter Process Communication similar to Electron. In contrast, Tauri forces developers from the beginning to use the Asynchronous Message Passing (AMP) paradigm, whereas Electron released this feature at version 14 and only recommends it. The main advantage of AMP is, that direct function access is denied and thus all communication has to go through the `core` process. Since it is able to observe the content of messages, it can decide which message will be forwarded and which will be blocked like malicious requests. Communication can be either unidirectional using events, which can be emitted by both `core` and `webview` processes to inform the event recipient but without any response, or using commands which are bidirectional IPC messages but can only be emitted by the `webview` processes to invoke functions that require access to the operating system.

3.1.2 Context Isolation

In contrast to Electron Tauri uses different patterns for isolating critical API calls communication between the `webview` processes and the `core` process. They are called Brownfield and Isolation Pattern, whereas the default pattern, that can be configured inside the `tauri.conf.json` is the Brownfield pattern

Brownfield Pattern:
The Brownfield pattern can be seen as a design pattern to ensure interoperability between new implemented and existing software. This pattern does not categorize software as legacy software but as current state of the art and software developed following this pattern tries to coexist and consider existing software as much as possible. This requires a deep knowledge of the existing software and also can result in re-developing significant parts of the existing software when tried to enhance new features. Tauri uses this pattern as standard and explains that it tries to be as compatible as possible. But unfortunately Tauri does not explain in detail how they are implementing this pattern and how this helps to avoid malicious frontend calls to the Tauri core [1].

Isolation Pattern:
The Isolation Pattern can be seen as an interposed instance between the IPC handler and the processes. This instance is providing a sandbox, called Isolation application, which is trusted and secure JavaScript code embedded into an `<iframe>`. The IPC handler passes its message to the Isolation application, where it is executed and may be modified or verified. After that it will be encrypted, passed back to the IPC handler and forwarded to the `core` process, where it will be handled as normal.

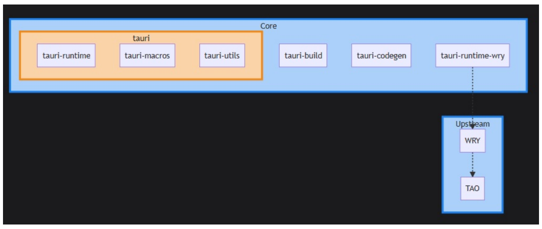

Figure 3: Core Ecosystem from [1]

As it can be seen in Figure 3 Tauri splits its framework into two main components. The Core, containing several modules providing API access, utilities and the runtime. The Upstream containing the two self-developed Rust crates WRY and TAO which are providing libraries for creation and rendering of WebViews and their content.

3.2 Frontend

As mentioned above, Tauris frontend relies on its own implementation of WebView instead of the entire Chromium content module for rendering HTML content. This makes it much smaller than electron applications since their libraries WRY and TAO are using the exising web engines of the three major operating systems Linux, macOS and Windows instead of shipping an entire browser.

TAO:
TAO is a cross-platform library written in Rust and used for creating and managing application windows. It was forked from the Rust crate `winit` since the developers wanted to enhance more desktop features than the original crate, like menu bars or a system tray. This library ships with an entire event loop that can be emitted by windows like resizing or key interactions and is included at the second frontend library WRY.

WRY:
WRY is the main library at Tauri application and responsible for rendering the content of the os-specific web view. It acts as an interface between the Tauri core with its low level webview drivers and the webview technology of the current operating system, providing an unique abstraction layer for rendering webview content. Since it re-exports TAO and its event loop to guarantee access to os-specific web engines the appearance of the same application may differ on each operating system, depending on the underlying web engine.

3.3 Backend

The backend or core of Tauri contains of several components, whereas some of them are summarized and called `tauri` to emphasize the main part of each Tauri application including runtime or the Tauri API. The decision to write the entire backend in Rust was made because of the ownership feature in Rust which provides a set of rules for memory management and avoiding a garbage collector like Java or explicit memory allocation like C. It can be seen as an approach of trying to be more comfortable for the developer as an explicit allocation but also to force the developer to consider his/her memory handling to implement an efficient app. This set of rules is checked at compile time and if one of them is violated, the program will not compile. Beside that `tauri` components there are also additional components included by the core like system-level interactions for the upstream crate WRY . One of the major advantages of using Rust for the Tauri backend is as already mentioned its ownership rules, avoiding security issues but also the execution time of Rust code which was intended to obtain a similar performance as C++. Furthermore, Rust experienced a significant attention resulting in the adoption of it by big companies like Google or Facebook which assumes an increasing position.

TAURI CORE ownership

4. IMPLEMENTATION OF A COUNTER APPLICATION

As already mentioned at 1.2 this section describes the underlying work for this paper and will guide through the entire development process of Electron and Desktop applications. To obtain comparable results, the methodology of comparison will be explained shortly. For this paper a basic counter application has been implemented in both Electron and Tauri using just bare HTML,CSS and JavaScript, although both frameworks are supporting most common frontend frameworks like Angular, React or Vue.js. This decision was made because the paper focuses on presenting the differences and similarities of each framework to the reader and thus using complete frameworks will both blow up the entire application and not concentrate on the essentials. The counter application consists of a simple Counter which is displayed and two buttons providing the possibility to increment or decrement the counter. Both buttons will have an event listener, that sends IPC messages to the backend and as response get the new calculated counter value which then will be displayed to show the entire communication chain of each framework. Therefore, each application will contain the same HTML content as it can be seen in Figure **??** that is displayed and only the API calls of the frameworks will differ, depending on their architecture. To avoid unnecessary distortion of measurements only the fundamentals of each framework are included without any additional helper libraries for better styling or further functionalities.

4.1 Methodology

For benchmarking the applications in case of build time GitHub Actions are used. Thus, each project has its own GitHub repository with a workflow.yaml defining the actions' workflow. Each Action will run on the three major operating systems macOS, Windows and Linux, set the prerequisites for the framework respectively and use the recommended build tool, `electron-forge` for Electron and the `tauri build` command for Tauri. This will result in three jobs for each project, whereas the build time for each framework on each os will be measured, since usual prerequisites are installed once and thus not measured. Time of execution is measured using hyperfine[4] which is a command line benchmark tool. Therefore, the executable of each framework will be started by command line, with hyperfine switched in front. To see the difference between a cached and uncached execution, the first run does not have any warmup actions and so prevent the operating system from loading

[4] `https://github.com/sharkdp/hyperfine`

the application into the filesystem cache. This results in following command

```
# hyperfine -runs 10 -warmup 2 'start <executable>
```

To gather meaningful data, hyperfine will run 10 instances of the executable to obtain a min-max range as well as a mean execution time.

For memory consumption the python library memory-profiler[5] will measure the memory consumption of each executable over a timespan of 60 seconds, enough to get from startup to idle state. This timespan will be set to 60 seconds, to allow interaction with the application as well as complete startup and idle state. Since the applications rely on multiprocess models and may also spawn children, these are recorded too, to gather trustful memory consumption measurements. To archive this measurement following memory profiler command is used

```
# mprof run -include-children -multiprocess -timeout
60 <executable>
```

It is important to notice that Tauri comes with a single executable file, whereas Electron ships with an installer which has to be executed first in order to get the actual executable application running. All Electron measurements will use this installed executable as foundation.

Except build time for different operating systems which will use GitHub Actions, the measurements of execution time, memory consumption or binary size are made with Windows 11. The implemented application is based on Electron Version 19 the newest stable release at time of writing this paper respectively Tauri version 1.0.2.

4.2 Development

The development process is following the best practices section of each framework [15][1] to avoid deprecated or inefficient calls. Implementation is done with usage of Visual Studio Code, a common, lightweight IDE commonly used for web development and also recommended by both Electron and Tauri.

4.2.1 Prerequisites

Since Electron mainly relies on Node.js, this as well as the actual Electron framework are the only dependencies that are needed to start developing. It should be noticed, that Electron is installed as dev dependency since packaged applications are shipped with an Electron binary and therefore are not needed to be defined as production dependency. Although they are not used, since the application is based on just HTML CSS and JavaScript, Electron provides several tools for generating a project with fundamental boilerplate code for most common web frameworks like Angular or React.

Tauri in contrast needs several more prerequisites to start developing which includes WebView2, the actual web engine for Windows that is used by Tauri as well as Rust, what on the other hand pre-conditions Microsoft Visual Studio C++ Build tools. Since Tauri consists of two subprojects, the Rust Core and the Frontend it provides a tool for scaffolding boilerplate projects using the most common web-frameworks as well as just plain HTML CSS and JavaScript (called Vanilla.js by Tauri), whereas Node.js as prerequisite is implied too. To initiate the Rust project, once the entire project is scaffolded, the CLI module of tauri needed. This tool will generate a minimal Rust project set-up to use Tauri with the selected web-framework and specify the location of all the belonging web assets.

Once each project is initialized a basic HTML file combined with basic CSS styling is created to specify the actual appearance of the application. The main JavaScript file of the frontend is specified inside the `package.json` of each framework. As mentioned before, two buttons will be added to increment or decrement the counter for presenting a basic IPC. Therefore, each button has an event listener which will listen to click events and call the appropriate IPC chain.

4.2.2 Implementation - Electron

Following best practices the Electron framework is divided into three JavaScript files, whereas the main process (`electron.js`) represents core process as described in section 2.1. This file contains the actual browser creation which will spawn a new renderer process with the corresponding preload script as well as specifications of the content that should be loaded inside the window. But also required handlers of the `ipcMain` module including the logic are implemented here. The second JavaScript is the preload script that contains the `contextBridge` and `ipcRenderer` modules of Electron and exposes the API to the renderer process. This preload script is linked to the main process at the definition of the browser window, that is created by the `electron.js` file. The third JavaScript file is the renderer script, that is responsible for adding basic interaction processing that can occur.

```
inc_btn.addEventListener('click', async ()
    => {
    const res = await window.electronApi.
        increment(document.getElementById(
        'counter').innerText)
    counter.innerText = res;
})
```

Listing 1: Excerpt of render.js

```
contextBridge.exposeInMainWorld('
    electronApi', {
    increment: (param) => ipcRenderer.
        invoke('incrementChannel', param),
})
```

Listing 2: Excerpt of preload.js

```
ipcMain.handle('incrementChannel', async(
    param) => {
    return parseInt(param) + 1;
})
```

Listing 3: Excerpt of electron.js

Once a button is pressed a click event will be emitted and the code shown at listing 1 will be executed. Inside the event listener the related function of the API, that is exposed by the preload script and shown at listing 2, will be called. This will invoke the `ipcRenderer` module which takes the params of the `increment` function and send a message using the defined channel, at this case `incrementChannel` and wait for

[5] `https://pypi.org/project/memory-profiler/`

the response of the handler of the `ipcMain` module at listing 3. Since the handler is inside the `electron.js` which is the main process of the application, this is the place where node modules can be imported and thus the only way for renderer process to use them. The handler will return the new calculated passed parameter back to the `ipcRenderer` which then will be set as new counter value as it can be seen at listing 1. Addressing the difficulty of implementing an application with Electron is has to be said that the entire app can be written with JavaScript,HTML and CSS which may have a great impact on small teams of web-developers that are already familiar with those web technologies. The framework itself is well documented including examples that cover of most use cases and constantly updated to follow the recommendations of the newest releases by the maintainers. Since Electron uses Node.js as runtime it benefits from its huge community providing libraries for almost any scenario. Nevertheless, since Electron has released up to 19 stable release versions, there has changed a lot over the years, causing developers to update their applications constantly, which in worst case could result in reimplementing the entire process communication to migrate their codebase.

4.2.3 Implementation - Tauri

Once the two subprojects of the Tauri application are initialized the same frontend as described in section 4.2.2 will be added to the specified web assets folder, containg the HTML file, a CSS stylesheet as well as the JavaScript file for processing interaction events, at this example the EventListener of both buttons.

```javascript
incBtn.addEventListener('click', function () {
  invoke('inc', {cnt: parseInt(counter.innerText)}).then((response) => counter.innerText=response)
})
```

Listing 4: Excerpt of index.js

```rust
#[tauri::command]
fn inc(mut cnt: i32) -> String {
  cnt+=1;
   return cnt.to_string();
}

fn main() {
  tauri::Builder::default()
    .invoke_handler(tauri::generate_handler![inc])
    .run(tauri::generate_context!())
    .expect("error while running tauri application");
}
```

Listing 5: Excerpt of main.rs

The core process as described in section 3.1 is created by the `main.rs` file of the Rust project inside the main function of listing 5. Inside the main method the default method of Tauris Builder structure is executed which will set WRY as runtime for the application. The `invoke_handler` method will set up the passed handlers of the application that will be responsible for IPC. Inside the `run` method, that will execute the passed context, the actual context is generated by the `generate_context` method, that will read the config file, which is the `tauri.conf.json` by default and create the context of the application. To indicate handlers that will be executed once the fronted makes an IPC request to the core, Tauri uses the `#[tauri::command]` makro for commands and `#[tauri::event]` for events respectively. As it can be seen at listing 4 once a button is pressed, the implemented event handler will be called. This will call the `invoke` method of the Tauri API which can either be imported as npm package or set to global inside the config file. The method takes two parameters whereas the first defines the name the command that should be executed, similar to the channels of Electron and the second parameter is containing all data that should be passed to the command method and is returning a promise which then can be processed. Since Tauri is using a protocol similar to JavaScript Object Notation - Remote Procedure Call (JSON-RPC) the passed data has to serializable. After the IPC request has reached the core, the corresponding method will be executed which can be seen in listing 5. The command takes integer as parameter, increments it and then returns a string, that will be wrapped into a promise. After the invoke method returned the promise which contains the processed data of type string it will update the counter. Referring the complexity and difficulty the reader should keep in mind that implementing a Tauri application nowadays requires knowledge with Rust, since the backend and therefore the process communication can only be implemented in Rust. Although the Tauri developers announced plans to provide the support of different languages for the backend at the time, this paper is written, Tauri only supports Rust as backend language. The documentation of Tauri tries to follow the same schema as Electron but is often indistinct and ambiguous or does not provide any detailed information of subjects. Especially the choice of using Rust as backend cuts limits the provided libraries to those of Rusts package manager cargo which limits the amount of usable libraries in contrast to Node.js and may result in writing own functionalities since.

4.3 Performance

Once the applications are implemented the build process of each framework is invoked to create an executable which is the foundation of performance measurements. The build time for each application is measured using GitHub actions, whereas it has to be mentioned, that this will be done on completely bare metal runners, that did not cache anything to obtain more comparable measurements since each framework may cache at different scales. After the executables are created the performance of both are measured using the techniques described in section 4.1.

4.3.1 Build Time

Table 4.3.1 summarizes the results of the GitHub Actions for each operating system. It is clearly visible that building a Tauri app requires much more time than Electron applications spread over the operating systems. This can be explained with the complex process of Rust compilation like incremental compilation techniques or time-consuming code analysis [16] combined with the absence of caching since the GitHub Actions were executed on bare-metal systems. This owed due the fact of providing comparable results and may not occur at every-day development where applications are build constantly to test features and therefore cached to reduce compile time.

Building time in seconds				
Framework	Operating System			Size [KB]
	Windows	Ubuntu	MacOS	
Tauri	863	616	389	6 924
Electron	115	121	44	145 361

The binary size of the compiled executable amounts to 6 924 for the Tauri application respectively 145 361 for Electron

4.3.2 *Memory Consumption*

Lore ipsum

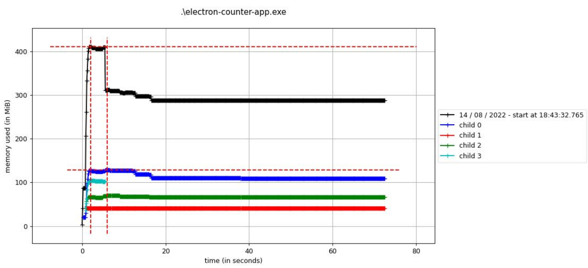

Figure 4: Memory consumption of Electron executable obtained from memory profiler

4.3.3 *Execution Time*

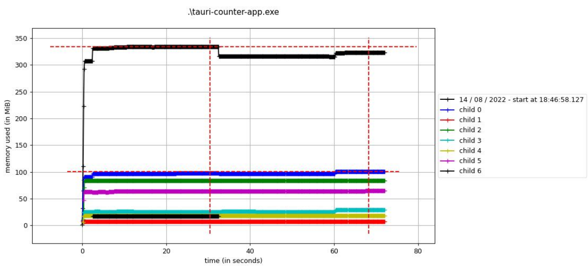

Figure 5: Memory consumption of Tauri executable obtained from memory profiler

4.3.4 *Execution Time*

The measurements of `hyperfine` are shown below, whereas the mean values include their standard deviation

Execution time of Electron [ms]			
	Mean	Min	Max
Without caching	17.3 ± 23.3	6.5	82.9
With caching	18.0 ± 14.8	7.5	49.6

Execution time of Tauri [ms]			
	Mean	Min	Max
Without caching	17.1 ± 10.5	5.8	35.4
With caching	15.1 ± 7.9	5.2	29.0

5. SUMMARY

5.1 Differences

5.2 Similarities

6. CONCLUSIONS

7. LIST OF ABBREVIATIONS

HTML Hypertext Markup Language

CSS Cascading Style Sheets

API Application Programming Interface

AMP Asynchronous Message Passing

JSON-RPC JavaScript Object Notation - Remote Procedure Call

IDE Integrated Development Environment

NPM Node Package Manager

IPC Inter-Process Communication

OS Operating System

DOM Document Object Module

IDE Integrated Development Environment

8. REFERENCES

[1] Tauri blog. `https://tauri.app`. [Accessed 30-July-2022].

[2] V8 documentation. `https://v8.dev/docs`. [Accessed 13-Aug-2022].

[3] A. Alkhars and W. Mahmoud. Cross-platform desktop development (javafx vs. electron).

[4] Leo Becker. Google-chrome-entwickler: Apple hält das web zurück. *heise online*, 2019. [Accessed 13-Aug-2022].

[5] Alexander E. Gorbalenya, Susan C. Baker, Ralph S. Baric, Raoul J. de Groot, Christian Drosten, Anastasia A. Gulyaeva, Bart L. Haagmans, Chris Lauber, Andrey M. Leontovich, Benjamin W. Neuman, Dmitry Penzar, Stanley Perlman, Leo L. M. Poon, Dmitry V. Samborskiy, Igor A. Sidorov, Isabel Sola, John Ziebuhr, and Coronaviridae Study Group of the International Committee on Taxonomy of Viruses. The species severe acute respiratory syndrome-related coronavirus: classifying 2019-ncov and naming it sars-cov-2. *Nature Microbiology*, 5(4):536–544, Apr 2020.

[6] Steve Kinney. *Electron in Action*.

[7] Elias Müller. Web technologies on the desktop: an early look at flutter, 2021.

[8] Lucas Fernandes Nogueira. v0.3.0 - tauri alpha. `https://github.com/tauri-apps/tauri/releases/tag/v0.3.0`, Dec 2019. [Accessed 13-Aug-2022].

[9] E.R. Pernice et al. Application Development in the Face of Evolving Web Technologies at the National Ignition Facility. In *Proc. ICALEPCS'19*, number 17 in International Conference on Accelerator and Large Experimental Physics Control Systems, pages 1052–1056. JACoW Publishing, Geneva, Switzerland, 08 2020. https://doi.org/10.18429/JACoW-ICALEPCS2019-WEMPR006.

[10] The Chromium Projects. Blink (rendering enginge). `https://www.chromium.org/blink/`. [Accessed 13-Aug-2022].

[11] Kevin Sawicki. Atom shell is now electron. `https://www.electronjs.org/blog/electron`, Apr 2015. [Accessed 26-May-2022].

[12] Gian Luca Scoccia and Marco Autili. Web frameworks for desktop apps: An exploratory study. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[13] `https://github.com/tauri-apps`. Tauri. `https://github.com/tauri-apps/tauri`. [Accessed 13-Aug-2022.

[14] `https://www.electronjs.org/`. Electron. `https://github.com/electron/electron`. [Accessed 30-Jul-2022.

[15] `https://www.electronjs.org/`. Electron documentation. `https://www.electronjs.org/docs/latest`. [Accessed 13-Aug-2022].

[16] David Wood. *Polymorphisation: Improving Rust compilation times through intelligent monomorphisation*. PhD thesis, MS thesis, School of Computing Science, University of Glasgow, Glasgow, Scotland, 2020.

[17] Cheng Zhao. Release v0.1.0: Update node: use node's implementation of set immediate. `https://github.com/electron/electron/releases/tag/v0.1.0`, Jul 2013. [Accessed 26-May-2022].