Table of Contents

List of figures

List of tables

Ⅰ. Introduction

These days, computer work has become so common that anyone uses a computer for their work. The problem is, actually lots of these works are repetitive and tedious. It could be opening a browser to check any new email arrived, finding their chrome search history, or clicking and capitalizing all words in some text according to their usage pattern. Many users waste their time doing unnecessary repetitive tasks on their computers.

Computer users usually have their own usage patterns. Those usage patterns could be referred to as a kind of workflow. If there is some automating application for their own workflows, they can reduce his/her boring work time efficiently.

For example, if some users may find chrome history more often than other users, and there is a kind of customized application for finding chrome history (chrome workflow), the user simply does the job through the workflow.

In macOS, there is an application named *Alfred* to make computer users do their work efficiently through customized workflow. Lots of macOS users use *Alfred* and many workflows for productivity, but *Alfred* is not free and runs only on macOS.

If there is a cross-platform workflow automating application, once a user makes their extension, they use the extension in other OSs, too. In addition, a uniform environment gives users more comfort. So, *Arvis* aims to be a cross-platform workflow automating launcher based on open source.

This document describes *Arvis* and related libraries' implementations. This document does not focus on *Arvis*' usage, feature, or extension development. (I made other resources about these including documentation homepage)

First, I will describe, why I'm started making *Arvis*, the differences with *Alfred*, related libraries, how they work together. And I will describe *Arvis*' GUI implementations at a high level. And then, I will describe differences with other related applications in Evaluation, Related works. And I will explain some problems of *Arvis*, and alternative of those problems in discussion. At last, I will summarize the paper's content in the conclusion.

II. Background

1) Terminology

* **Alfred:** Alfred is an launcher for macOS which boosts efficiency with hotkeys, keywords, text expansion and more

* **Alfred-workflow:** Alfred workflows allow users to enhance Alfred's power and do things that can never be possible with Spotlight

* **Alfy:** Alfy is a library for alfred users to create and distribute their alfred workflow simply.

* **Electron:** The Electron framework lets web developers write cross-platform desktop applications using JavaScript, HTML and CSS. It is based on Node.js and Chromium.

* **Packal:** Packal is a dynamic repository for Alfred Workflows and Themes.

* **Github:** Website and cloud-based service that helps developers store and manage their code, as well as track and control changes to their code.

* **Chromium:** Open-source browser project that aims to build a safer, faster, and more stable way for all users to experience the web.

* **Node.js:** JavaScript runtime built on Chrome's V8 JavaScript engine.

* **Github action:** GitHub Actions makes it easy to automate all your software workflows, now with world-class CI/CD. Build, test, and deploy your code right from GitHub.

* **react:** React (also known as React.js or ReactJS) is a free and open-source front-end JavaScript library for building user interfaces or UI components. It is maintained by Facebook and a community of individual developers and companies.

* **json:** JSON (JavaScript Object Notation, is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values)

* **plist:** Property lists are files for serializing objects used in OS X, iOS, NeXTSTEP, and GNUstep programming software frameworks

* **debouncing:** Debouncing is a programming practice used to ensure that time-consuming tasks do not fire so often, that it stalls the performance of the web page. In other words, it limits the rate at which a function gets invoked.

* **npm:** npm is the world's largest software registry. Open-source developers from every continent use npm to share and borrow packages, and many organizations use npm to manage private development as well.

* **scriptfilter:** The Script Filter input is one of the most powerful workflow objects, allowing users to populate Alfred (or arvis)'s results with their own custom items through a custom script or executable binary file.

* **webpack:** Webpack is a module bundler. Its main purpose is to bundle JavaScript files for usage in a browser, yet it is also capable of transforming, bundling, or packaging just about any resource or asset.

* **babel:** Babel is a tool that helps you write code in the latest version of JavaScript.

* **base64:** A group of binary-to-text encoding schemes that represent binary data (more specifically, a sequence of 8-bit bytes) in an ASCII string format by translating the data into a radix-64 representation

* **CLI (Command line interface):** A Command Line Interface connects a user to a computer program or operating system. Through the CLI, users interact with a system or application by typing in text (commands). The command is typed on a specific line following a visual prompt from the compute

* **redux:** Predictable state container for JavaScript apps. It helps developers write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

* **chrome devtools:** A set of web developer tools built directly into the Google Chrome browser.

* **IPC:** An abbreviation for inter-process communication (IPC). IPC is a mechanism that allows processes to communicate with each other.‘

* **Extension:** In this paper, Extension means a generic term for arvis-workflow and arvis-plugin.

2) Motivation

Lots of alfred-workflows can work on cross-platform but cannot because *Alfred* runs only in macOS. cross-platform users may want to use *Alfred*'s workflow when they work on other OSs other than macOS.

It could be a little complex to find useful alfred workflows. Lots of workflows just exist on *Github* without enough promotion. There is a workflow collection homepage called *packal,* but *packal* is zombie, hasn't been maintained for a long time, isn't familiar with many people, and not all workflows are uploaded in the packal.

More powerful interacting quicklook supporting html, pdf, markdown, normal text, and more can be very helpful. And because electron loads full feature of *chromium*, implementing more powerful quicklook in cross-platform is affordable.

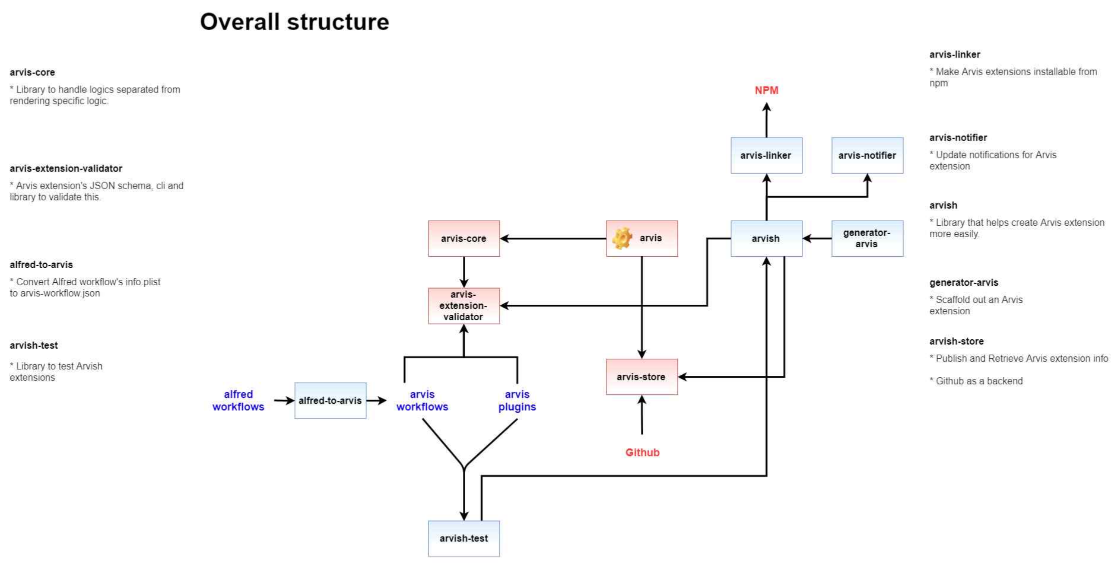
In *Alfred*, when a user uses some extension, the user must type some command. In some kind of extension, this process might mean just a waste of time. For example, when a user would like to find some file through an extension, the user cannot find the file by only typing the file name. The user must type some command to execute the command. Finding files through just file name is not attainable through *Alfred* workflow, so *Alfred* provides some built-in features like file searching, calculating, and other necessary utilities. Making this process other kinds of extension could improve the extensibility of user extensions.

To sum up, *Arvis* supports the following features to resolve these problems.

1. Available on cross-platform (Linux, macOS, Windows)
2. More simpler extension installation through GUI store feature
3. Advanced quicklook
4. Support more extensible extension additionally.

III. Architecture

1. Overall structure



Overall structure
arvis-core
* Library to handle logics separated from rendering specific logic.
arvis-extension-validator
* Arvis extension's JSON schema, cli and library to validate this.
alfred-to-arvis
* Convert Alfred workflow's info.plist to arvis-workflow.json
arvish-test
* Library to test Arvish extensions
arvis-linker
* Make Arvis extensions installable from npm
arvis-notifier
* Update notifications for Arvis extension
arvish
* Library that helps create Arvis extension more easily.
generator-arvis
* Scaffold out an Arvis extension
arvish-store
* Publish and Retrieve Arvis extension info
* Github as a backend
NPM
arvis-linker
arvis-notifier
arvis-core
arvis
arvish
generator-arvis
arvis-extension-validator
arvis-store
alfred workflows
alfred-to-arvis
arvis workflows
arvis plugins
Github
arvish-test

<Figure 1> Overall structure

'Figure 1' represents overall structure of *Arvis* and related libraries.

Red rectangles represent *Arvis* GUI and libraries in *Arvis* GUI. Blue rectangles represent *Arvis* extension related libraries. Red letters represent servers arvis extensions are hosting. Blue letters represent extension data existing as file form. Straight lines between rectangles represent the dependency relationship. For example, *Arvis* depends on *arvis-core*, *arvis-store*. Straight lines between rectangle and blue letter represent the package use the data. For example, *alfred-to-arvis* use alfred-workflows as input, use arvis-workflow as output. Straight lines between a rectangle and a red letter represent data flow from outside servers. For example, *arvis-store* fetches data from *Github* repository, *arvis-linker* uploads arvis extension to npm.

Overall, *Arvis* loads *arvis-core* from the search window to search for installed arvis-workflows whenever the user's search string changes, and it runs all arvis-plugins to display the items returned as a result in the search window.

Created extensions can be tested with *arvish-test*, uploaded to the *arvis-store*, and new extensions can be created using *arvish*. *arvish* can upload an extension at npm via *arvis-linker*, and *arvis-notifier* can inform users of some extensions are outdated. Extension developers can also create a skeleton project for extension projects through *generator-arvis*. The next section describes these libraries' detail in Figure 1.

2. Related libraries

Lots of alfred-workflow are built with *alfy.* This means with *alfred-to-arvis* (converter), *Arvis* users can convert lots of alfred-workflows to arvis-workflows if there is *alfy* of *Arvis*. So, I cloned *arvish* and arvish-workflows from *alfy*, and *alfy* workflows.

Because changing *arvish*'s API causes incompatibility with *alfy, arvish*'s API (on workflow) is basically totally same with *alfy.* And *arvish* has other cloned libraries that *alfy* use, too. (*arvis-linker*, *arvis-notifier,* and the others)

1) arvish

arvish provides CLI tools and API for arvis-extension to arvis-extension developers. *arvish*'s API is almost same with *alfy*'s to keep compatibility with *alfy* workflows.

2) arvis-linker

arvis-linker makes *Arvis* extensions installable from npm.

3) arvis-notifier

arvis-notifier checks arvis-extension's latest version from npm. and if the user extension's version is outdated, mark this on the extension's *arvis-workflow.json* (or *arvis-plugin.json*). *Arvis* checks each extension's latest version on *arvis-workflow.json* when *Arvis*' startup, if there are any outdated extensions, notify the user to update him (her) out-of-dated extensions.

4) generator-arvis

Extension developer can generate a skeleton file of *arvis-workflow.json* file, *arvis-plugin.json* file through *arvish*. But users also can generate a whole project of the extension through *generator-arvis*. *generator-arvis* is built with *yeoman*, well-known scaffolding tool. With *generator-arvis*, an extension developer can scaffold out *Arvis* extension project's skeleton files easily.

5) arvish-test

arvish-test is a test library for arvis-extensions. arvis-extensions needs *Arvis* environment variables to execute. So, arvis-extension cannot run outside of *Arvis*. *arvish-test* provides these environment variables instead of *Arvis* for testing.

6) arvis-core

arvis-core handles logics separated from rendering specific logics.

7) alfred-to-arvis

alfred-to-arvis works in the following sequence.

1. Read *info.plist* and iterate all node, find trigger node (which means input type is hotkey or keyword or scriptfilter.)
2. Repeat finding next nodes starting with the trigger node recursively.
3. If next node is not supported type, stop iterating, find another next node.
4. If next node is supported, extract the node's information.
5. Write extracted information to *arvis-workflow.json*

8) arvis-extension-validator

arvis-extension-validator includes json-schema for validating arvis-extension and also includes a library for validating. *Arvis* use the library when installing new extensions to filter out invalid extensions which might break *Arvis*'s behavior. Extension developers also can use *arvis-extension-validator* CLI tool for validating their extension. (The CLI tool is also included in *arvish*)

And by including this library's json schema in their extension json file, vscode automatically checks if the extension json file is valid. *arvish*'s '*init'* command and *generator-arvis* makes a skeleton extension json file including this library's schema.

9) arvis-store

arvis-store works as '*Github* as a backend', which means all data used in *arvis-store* are stored in *Github*, and these data are updated through *Github* action.

The *Github* action for updating store is triggered every 12 hours automatically. Whenever the action is triggered, '*internal/store.json*' file is updated. This file store each extension's weekly, total download count, name, latest version.

By doing this update, each extension's info can be kept up to date.

To add a new extension to *arvis-store*, extension developers need to register their Github authentication token to *arvis-store* in a CLI tool. With this token, *arvis-store* CLI tool can make PR (pull request) adding the extension info to '*internal/static-store.json*'. The static-store.json stores the extension's static information like name, creator, description, homepage.

This information usually needs not to be updated periodically (If they need the renewal of extension info, they can manually create a PR editing the static info.) And *arvis-store* provides APIs letting users get extension information. *Arvis* use the APIs for providing GUI store feature in the preference window.

There are several types that extension developers can upload. These types are specified when extension developer upload their extension. The types indicate how the extension should be installed.

* npm: Install the extension from npm.

* local: Download the extension data as base64 format from *arvis-store* and install that by local.

3. *Arvis* GUI architecture

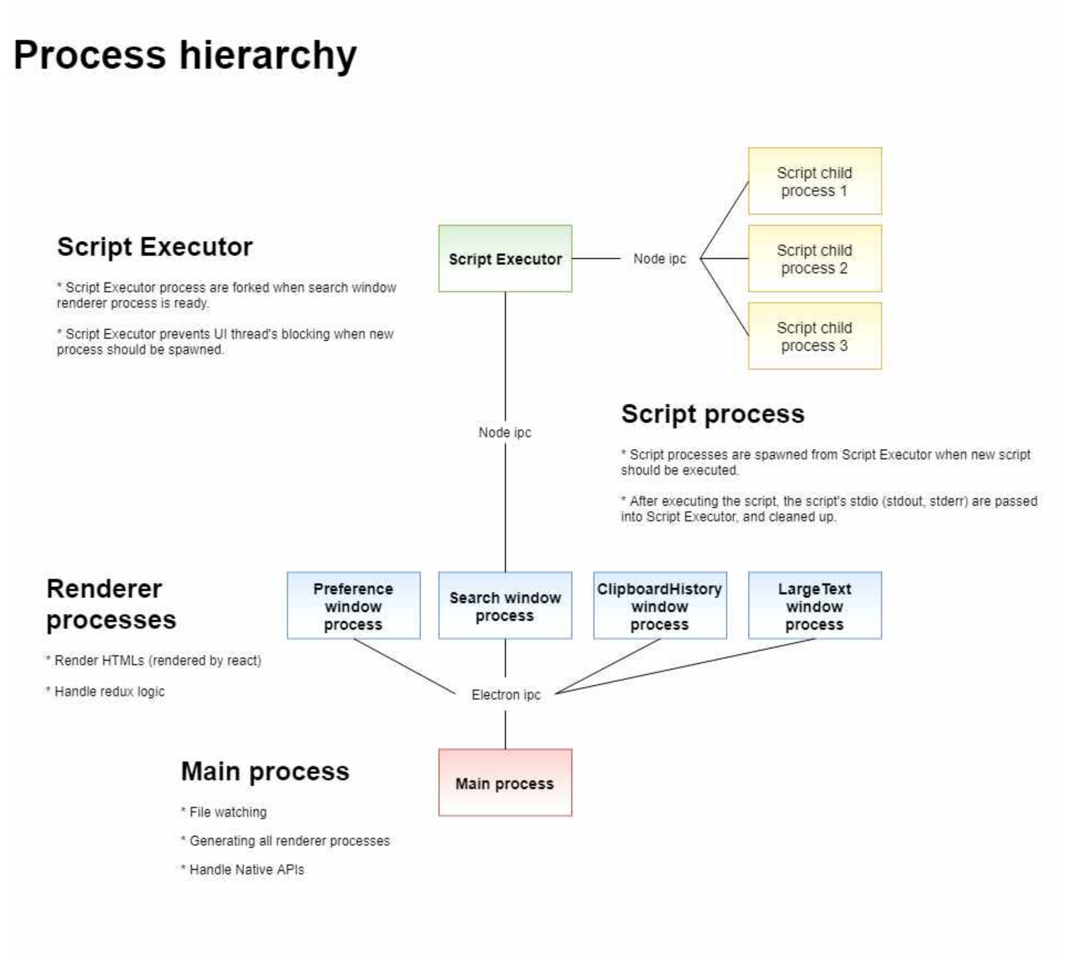
1) Electron

Arvis chose *Electron* as a GUI framework for the following reasons. *Electron* supports the full feature of *chromium*. This means lots of features of *chromium* are available with minimal effort.

Electron also has lots of awesome libraries to boost developers' productivity, it enables developers to do more works in the same time.

Many alfred-workflow are built with *alfy* (Built with javascript). *Arvis* also needs library help users to create their own extensions easily. So, I decided to use clone *alfy* and related libraries. So, choosing *Electron* as a framework also means I can unify programming language with javascript in whole architecture. It also helps to improve overall productivity.

2) Process Hierarchy


Process hierarchy
Script Executor
* Script Executor process are forked when search window renderer process is ready.
* Script Executor prevents UI thread's blocking when new process should be spawned.
Script Executor
Node ipc
Script child process 1
Script child process 2
Script child process 3
Node ipc
Script process
* Script processes are spawned from Script Executor when new script should be executed.
* After executing the script, the script's stdio (stdout, stderr) are passed into Script Executor, and cleaned up.
Renderer processes
* Render HTMLs (rendered by react)
* Handle redux logic
Preference window process
Search window process
ClipboardHistory window process
LargeText window process
Electron ipc
Main process
* File watching
* Generating all renderer processes
* Handle Native APIs
Main process

<Figure 2> Process hierarchy

Figure 2 represents *Arvis* GUI’s process hierarchy. Two nodes are connected denotes below process is a parent process of the other process in the figure.

Electron runs on the main process. The main process spawns renderer processes. Search window renderer process spawned script executor process with node ipc channel when *Arvis* starts up.

Whenever the script executor runs some script, a new child process spawned from the script executor process and previous (other) script executor’s child processes are canceled and killed.

Ⅲ. Arvis GUI Implementation details

1. Module structure

Arvis is built with *electron-react-boilerplate*. So, basically, project structure and *Webpack*, *Babel* setting files are from the *electron-react-boilerplate*.

# Two package.json structure

'*./src/package.json*' relative to the project root
: native module dependencies should be specified in this *package.json*
For example, *iohook*, *fsevents*, *robotjs* are specified here because they are native modules which means they are built according to the platform at build time.

'*./package.json*' in the root of your project
: All modules except for native modules are specified in this *package.json*

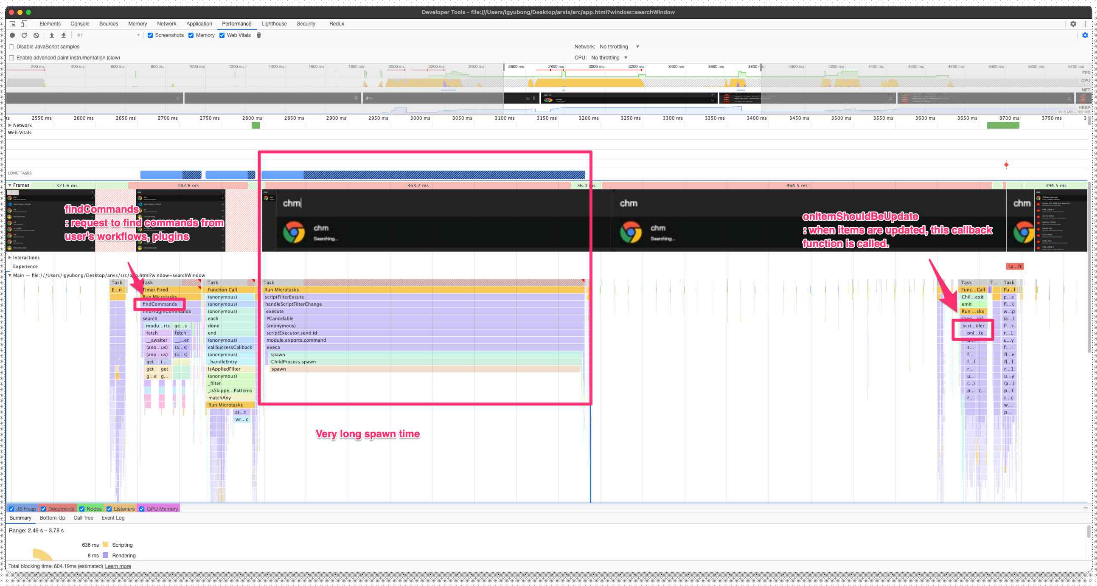
# External modules

External modules are '*src/external*'. which are the following.
* Type declarations of type-less modules (*easy-auto-launch*)
* Libraries with extended functionality (*use-key-capture*)
* Libraries with no webpack-bundling (*about-window*)
* css files for updating existing module's style
(*react-tabs, react-prosidebar, github-markdown-css, jsoneditor*)

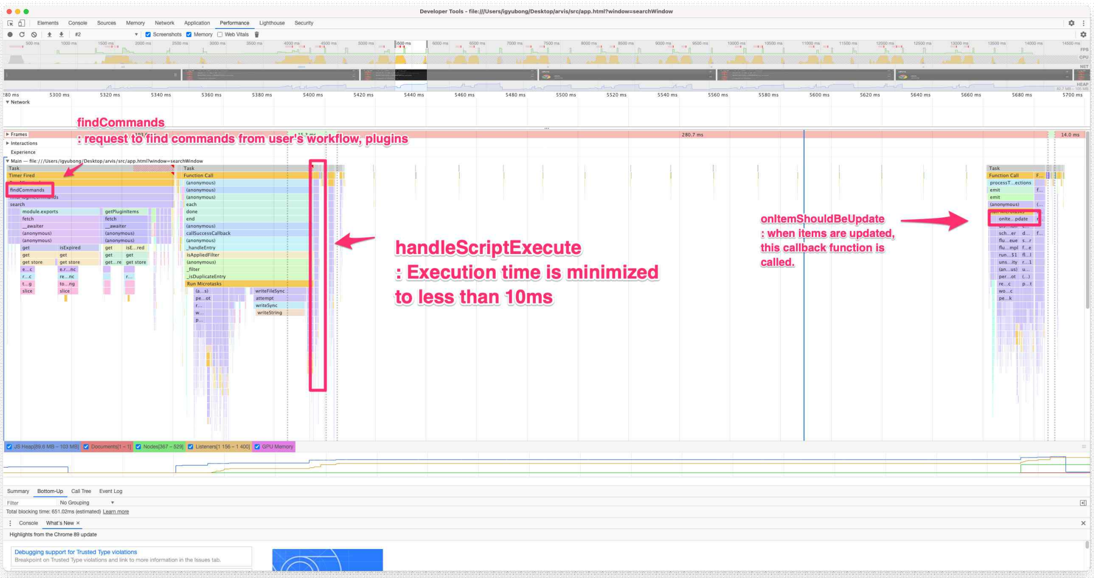
2. Script executor

Script executor process exists for removing process spawning time from search window process when a new script should execute. Without script executor process, process spawning time delays renderer process. This leads to the result of the main process's UI thread blocking whenever the users type something in scriptfilter.

Especially, in the macOS Bigsur, the delay time exceeds 150ms which means scriptfilter is barely available to the macOS users without script executor process. Figure 3 denotes there is a very long spawning time between 'findCommands' function and 'onItemShouldBeUpdate'



<Figure 3> Process spawning without Script Executor process


findCommands
: request to find commands from user's workflow, plugins
handleScriptExecute
: Execution time is minimized
to less than 10ms
onItemShouldBeUpdate
: when items are updated,
this callback function is
called.

<Figure 4> Process spawning with Script Executor process

'handleScriptExecute' is a function name that included process spawn logic. Figure 4 represents this 'handleScriptExecute' function's execution time is minimized from 363ms to less than 10 ms.

3. Hotkey implementations

Hotkey implementation is a little tricky. For hotkey implementation, below three libraries are used.

* *iohook*
* *use-key-capture*
* *electron/globalshortcut*

# How to record key in hotkey record form

Key recording forms are worked with *iohook*. *iohook* starts at renderer processes when *Arvis* starts up. (*iohook* should be in renderer processes because it can cause unexpected SIGILL error when in the main process.)

When the hotkey record form is mounted, keydown event handler of *iohook* is registered. and when the form is unmounted, the handler is unregistered.

Because *iohook* keydown event handler only returns keycodes, *Arvis* translate these keycodes to a hotkey string before recording. For this transformation, *Arvis* uses '*utils/iohook/keyTbl*', '*utils/iohook/keyUtils*'.

In '*utils/iohook/keyTbl*', there is a table that shows matching key strings for all available keycodes.

In '*utils/iohook/keyUtils*', there are some utility functions for translating keycode and hotkey string.

# Double key handling

Arvis supports modifier double key recognition. For the implementation of this, *Arvis* use '*hooks/utils/doubleKeyUtils'*.

# How hotkey is registered (not double key)

Except for the double key, the other hotkeys are registered in *electron/globalshortcut*. This process occurs after *Arvis* start or some hotkeys are changed by the user.

# How double hotkeys works

Registering double hotkeys is not supported in *electron/globalshortcut*, *iohook* both. So *iohook* watches all modifier key press in keydown event handler, uses timers for each modifier key and recognizes double keypress when modifier key is pressed twice in a very short time.

For this implementation, *Arvis* uses '*hooks/utils/useDoubleHotkey'*. Because *iohook* should exist in the renderer process, double keypress handlers exist in the renderer process.

On the other hand, *electron/globalshortcut* should execute in the main process. So, after acquiring registered hotkey information (from *arvis-core*, redux-store) in the renderer process, double keys are registered in the renderer process, other key information are sent to the main process through ipc channel.

# How renderer process's hotkeys work

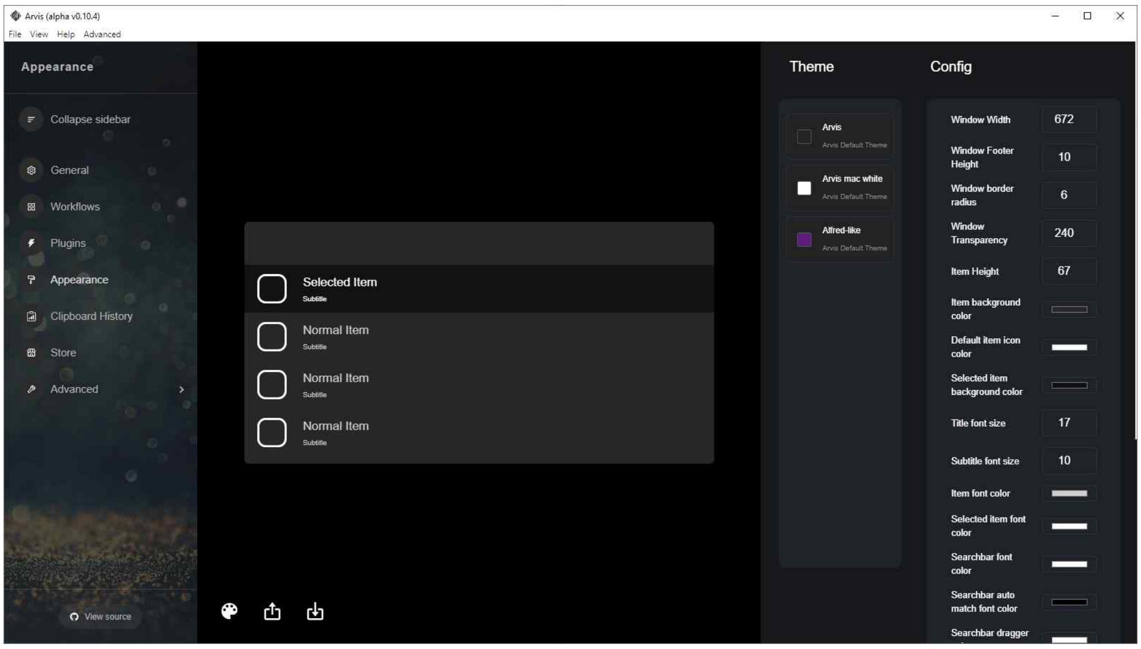
Some hotkeys should execute only in the renderer processes. For example, opening the large text window only be possible when the user focuses on the search window's items. Even if a user presses the hotkey, the large text window should not show up when the user does not focus on an item of the search window. So these hotkeys are not registered as separate hotkeys. Instead, these hotkeys are handled with *use-key-capture*'s key-down event handlers.

4. Appearance

Most variables used in UI styling are stored in *redux*, and the variables can be changed by users in the Preference window's Appearance page.

When users change the values, these changes are dispatched to the search window too, so the users don't need to restart *Arvis* to see what's changed.

The users also can export and import these GUI setting values through *arvistheme* file.



Arvis	Arvis Default Theme
Arvis mac white	Arvis Default Theme
Alfred-like	Arvis Default Theme

Window Width	672
Window Footer Height	10
Window border radius	6
Window Transparency	240
Item Height	67
Item background color	
Default item icon color	
Selected item background color	
Title font size	17
Subtitle font size	10
Item font color	
Selected item font color	
Searchbar font color	
Searchbar auto match font color	
Searchbar dragger	

<Figure 5> Appearance page

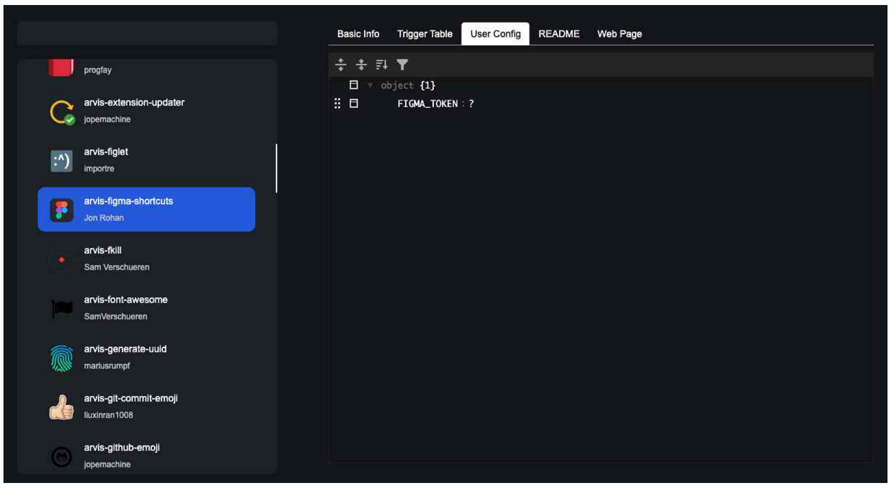
5. User Config

Alfred supports setting workflow environment variables, and these variables are stored *info.plist*. Because the variables are stored in *info.plist*, workflow developers can register necessary variables (like API key) on that tab. Then, workflow users can simply edit their own config values through the table.

Arvis also supports setting variables on the *arvis-workflow.json* config file. But *Arvis*'s variable values are stored as a separate file (named user-config file) at the same time. After the extension update, existing setting values are updated by the user-config file.

So extension developers can specify necessary variables to users, And extension users simply change these variables on user-config table of their preference window.

user-config file updated automatically whenever the user changes their setting value on user-config table.



```
object {1}
    FIGMA_TOKEN : ?
```

<Figure 6> User config

6. File watcher

Arvis use *chokidar* to keep watch file change event. File watching is used for extension auto reload.

For example, if a new *arvis-workflow.json* file is created or edited in the extension folder, *Arvis* detects the change, triggers workflow-reload function. By doing this, after installing new extensions, *Arvis* can reload its extensions, and always use the latest extensions.

On the workflow extension folders, *chokidar* only watches *arvis-workflow.json* because other files are not loaded in *Arvis*. (Other files are used through inter process communication)

But in the case of plugin, *chokidar* watches the extension root folder's javascript files, because they are all loaded in *Arvis*.

The reason *chokidar* doesn't watch all javascript files is that to reduce system resource consumption. *chokidar*'s file watching can use lots of system resources if *chokidar* should watch all files in extension folders.

Ideally, *chokidar* should watch all javascript files of custom plugins, but reducing system resources is more important than auto plugin-reload.

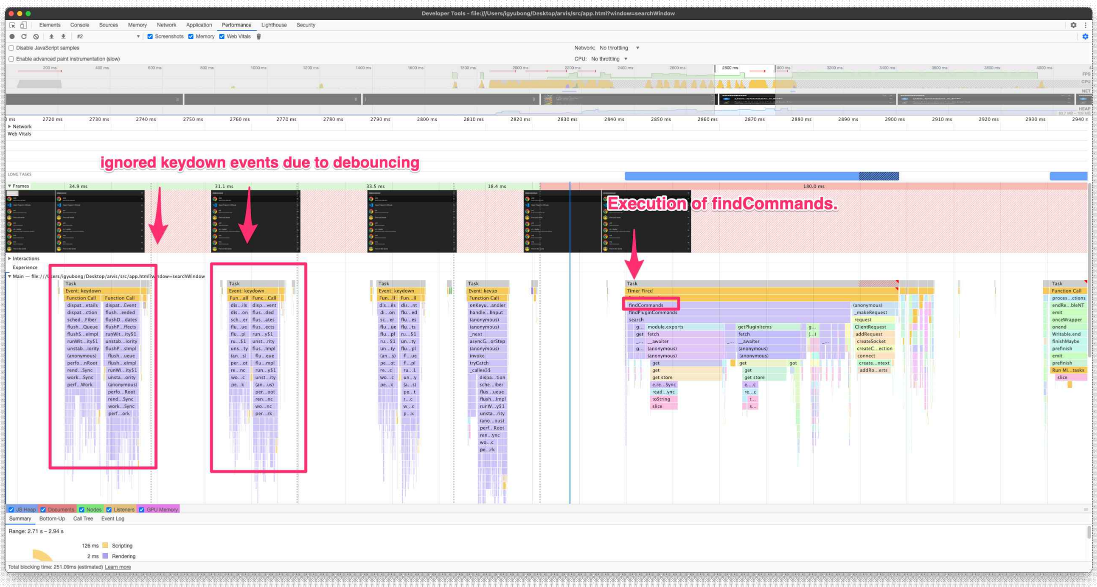
So *Arvis* restricts monitoring to the root folder's javascript files (and *arvis-plugin.json* file) only.

7. Debouncing optimization

In the search window, debouncing is applied to script executing with 25ms which means if several script executing requests were made within 25ms, only the final request execute and all other requests are ignored.

Reducing the number of executing requests helps to improve searching performance because pressing key consecutively is very common in launcher applications.

There is 25ms between the last '*keyup*' event handler and '*findCommand*' function, and the other keydown event handlers are all ignored because they are consecutive in 25ms in Figure 7.


ignored keydown events due to debouncing
Execution of findCommands.

<Figure 7> Optimization through debouncing

8. Global state management

1) Redux

Arvis stores following states in *redux*.
* Some global necessary hotkey
* Auto launch config
* Customizable UI config values
* Debugging setting (which type of logs should be logged in *chrome-devtool*)
* Clipboard history

On the other hand, *Arvis* doesn't consider if the *search window is pinning*, *clipboard is pinning* should be saved because *Arvis* consider the pin state should be off default. *Arvis* also does not store dynamically changing UI states (like spinner state) in *redux*. Saving these states doesn't make sense because it would be meaningless next time.

And *Arvis* does not store the extension's information. These pieces of information could be too massive to save in one file if the user uses a lot of extensions, and Extension's information does not need to be states. So it's just loaded on memory as forms of record object by *arvis-core*.

2) Dispatching action to other renderer processes

redux-store should be available on every renderer process. To maintain every process's redux-state up to date, *Arvis* use '*ipc/mainProcessEventHandler/config/dispatchAction.ts'*. This event handler is registered in the main process.

If any *redux* state is changed in some renderer process, the renderer process notifies the main process the state change through electron's ipc module. Then the main process sends the action to the destination renderer process.

3) How to persist redux state

To persist *redux* states, *Arvis* uses *redux-persist* and *redux-persist-electron-storage*. The redux states are stored as json file named *arvis-gui-config*, and loaded every time *Arvis* starts up.

4) How to handle upgrade redux state's schema

Arvis validates *redux* states by comparing '*config/initialState.ts'* with the current store's states at startup. If there are any differences, *Arvis* is forced to be terminated to reset the *redux* store's state. *redux* store's state reset is caused by replacing *arvis-gui-config* file.

By doing proper replacing *arvis-gui-config*'s key-values, the states change after *Arvis* restarts. At this time, user's customized values should not be changed. For this, *Arvis* makes *arvis-redux-store-reset* file to write the user's custom setting values before restart.

9. *arvis-workflow.json*

This section describes the reasons *Arvis* does not use *info.plist* of *Alfred* workflows. (*info.plist* refers to the config file of alfred workflow in this section.)

info.plist is not human-readable. *info.plist* has kind of DAG (Directed Acyclic Graph), this makes the *info.plist* not human-readable. So *Arvis* use an exclusive json format to replace the *info.plist*. The json format replaces the DAG structure with a simple tree structure made up of Trigger and Action, and removes each node's position information.

And the *info.plist* belongs to *Alfred*. it can be updated or changed without notice anytime, so using this in the *Arvis* is not reasonable.

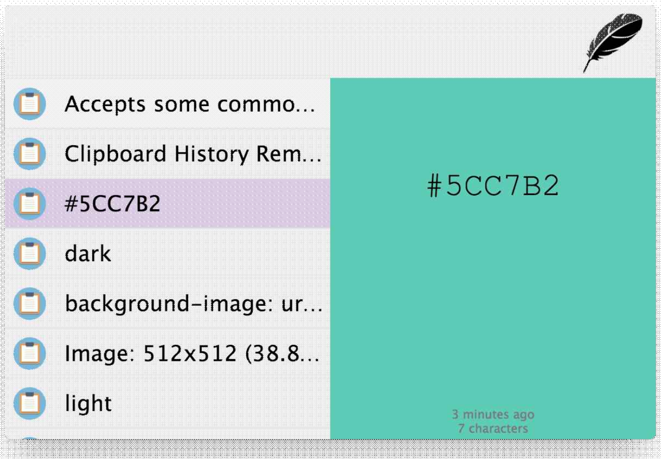
IV. Evaluation

Table-1 shows a feature comparison table between extensible launchers.

<Table-1> Feature comparison table

	Arvis	***Alfred***	**Wox**	**Dext**	**Cerebro**	**Zazu**
1. Can extension be written in any languages?	O	O	X (C#, python)	X (js)	X (js)	X (js)
2. Can UI be customized?	O	O	O	X	△ (Only have dark theme)	O
3. Can users simply install extensions both through GUI and CUI?	O	O	O	O	△	△
4. Do have lots of extensions? (50 or more)	O	O	O	△	O	O
5. Is fully supported cross-platform (Windows, macOS, Linux)	O	X (macOS)	X (Windows)	X (macOS)	O	O
6. Is currently actively maintained?	O	O	X	X	△	X
7. Is it compatible with *Alfred* workflow?	O	O	X	O	X	X
8. Support *clipboard history*?	O	O	X	X	X	X
9. Is open source and completely free?	O	X	O	O	O	O
10. Support *snippet* feature (Text auto expanding)?	O	O	X	X	X	X

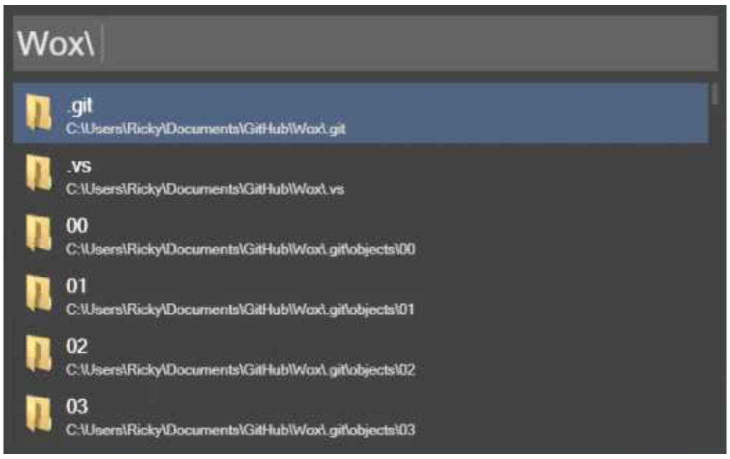
Wox runs only in Windows, *Wox* hasn't been maintained at the moment. Dext has compatibility with *alfy* workflow, but runs only on macOS. *Cerebro* supports Linux, macOS, Windows. But doesn't compatible with *Alfred* workflows, and it's been a long time since the last release. *Zazu* is also been a long time since the last release. And in common, these launchers do not support *clipboard history* and *snippet*.



```
Accepts some commo...
Clipboard History Rem...
#5CC7B2
dark
background-image: ur...
Image: 512x512 (38.8...
light
```

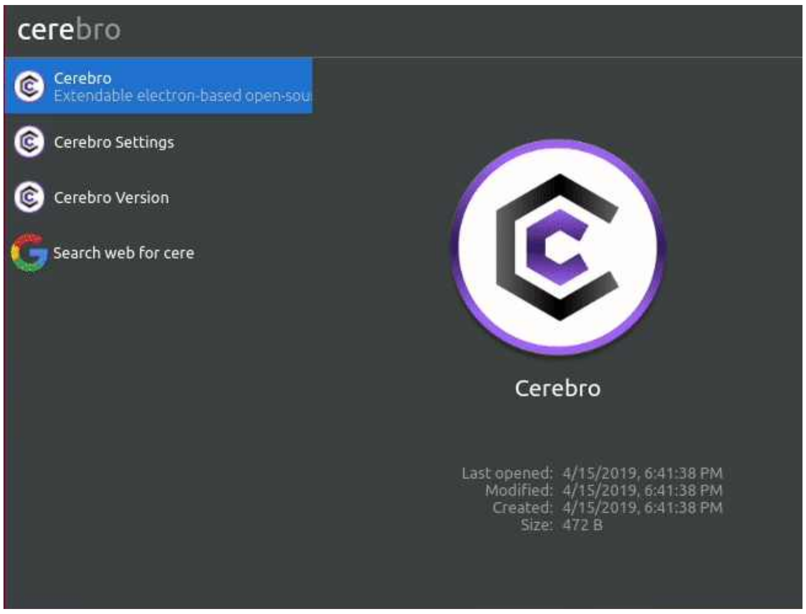
```
#5CC7B2

3 minutes ago
7 characters
```

<Figure 8> Zazu



```
Wox\
.git
C:\Users\Ricky\Documents\GitHub\Wox\.git
.vs
C:\Users\Ricky\Documents\GitHub\Wox\.vs
00
C:\Users\Ricky\Documents\GitHub\Wox\.git\objects\00
01
C:\Users\Ricky\Documents\GitHub\Wox\.git\objects\01
02
C:\Users\Ricky\Documents\GitHub\Wox\.git\objects\02
03
C:\Users\Ricky\Documents\GitHub\Wox\.git\objects\03
```

<Figure 9> Wox



cerebro

Cerebro
Extendable electron-based open-sou

Cerebro Settings

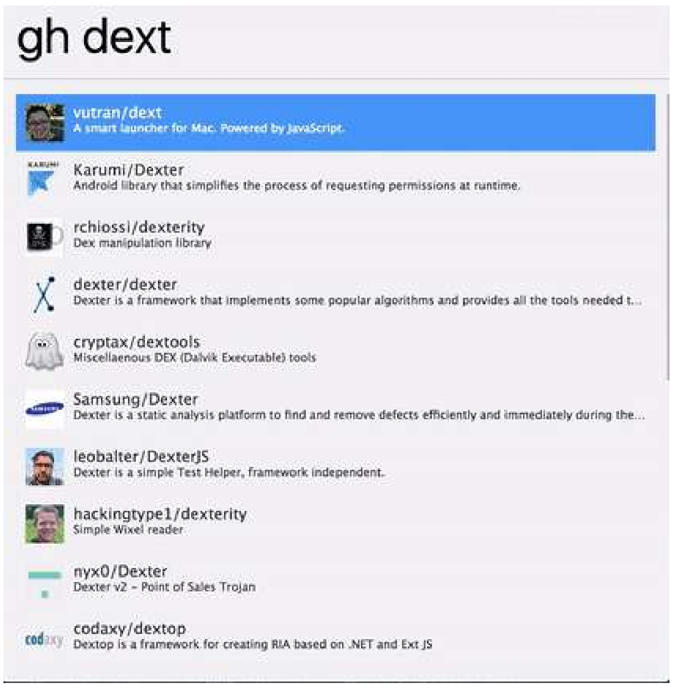
Cerebro Version

Search web for cere

Cerebro

Last opened: 4/15/2019, 6:41:38 PM
Modified: 4/15/2019, 6:41:38 PM
Created: 4/15/2019, 6:41:38 PM
Size: 472 B

<Figure 10> Cerebro



gh dext

vutran/dext
A smart launcher for Mac. Powered by JavaScript.

Karumi/Dexter
Android library that simplifies the process of requesting permissions at runtime.

rchiossi/dexterity
Dex manipulation library

dexter/dexter
Dexter is a framework that implements some popular algorithms and provides all the tools needed t...

cryptax/dextools
Miscellaenous DEX (Dalvik Executable) tools

Samsung/Dexter
Dexter is a static analysis platform to find and remove defects efficiently and immediately during the...

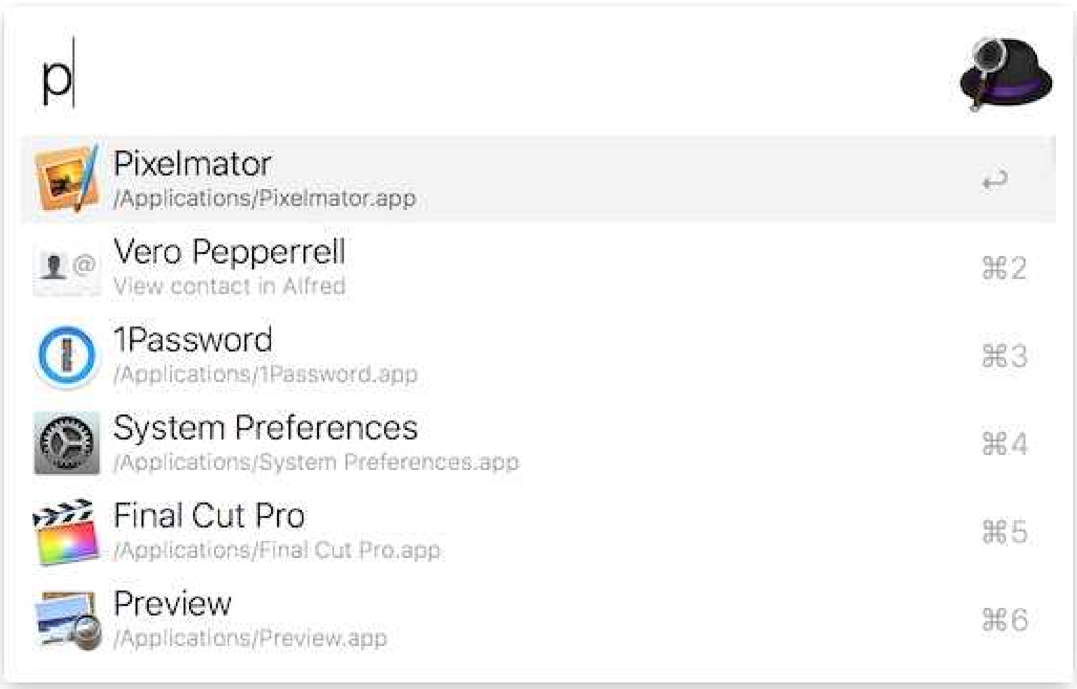
leobalter/DexterJS
Dexter is a simple Test Helper, framework independent.

hackingtype1/dexterity
Simple Wixel reader

nyx0/Dexter
Dexter v2 - Point of Sales Trojan

codaxy/dextop
Dextop is a framework for creating RIA based on .NET and Ext JS

<Figure 11> Dext



p

Pixelmator
/Applications/Pixelmator.app

Vero Pepperrell
View contact in Alfred ⌘2

1Password
/Applications/1Password.app ⌘3

System Preferences
/Applications/System Preferences.app ⌘4

Final Cut Pro
/Applications/Final Cut Pro.app ⌘5

Preview
/Applications/Preview.app ⌘6

<Figure 12> *Alfred*

V. Related works

As described in Table-1, *Alfred, Wox, Dext, Cerebro, Zazu* are all kinds of extensible launcher applications. Their differences are described in section IV, Evaluation.

automator is a comprehensive productivity tool to control the workflow of macOS. Compared with *automator*, *Arvis* provides a more easy and user-friendly interface. Above all, *automator's* custom workflows only can be written by *Applescript* (or *JXA*), which is slow and has fewer users compared to other widely used programming languages. And runs only on macOS.

VI. Discussion

1. *Arvis* extension auto reload doesn't work with symbolic link directory

: Because *chokidar* has an issue with symlink currently, *chokidar* doesn't follow a symbolic link directory, so auto reload feature does not work with the symbolic link directory. This issue will be resolved after *chokidar's* symbolic link issue is resolved.

2. *Arvis* is freezing at startup for a while

: It might be because heavy operations block UI thread currently at startup. It seems this issue may be resolved by moving this async operation to the other process, or web worker.

3. *Arvis* uses too much memory

: Because *Electron* fully loads *chromium*, *Electron* app inevitably uses lots of memory. Additionally, *Arvis* uses more memory to render the complex preference window, assistance window, and others compared with other launchers. It appears to be the limitation of *Arvis*.

VII. Conclusion

Computer users usually have their own usage patterns. To boost users' work productivity, users need a kind of more advanced extensible launcher application. Existing available launchers have several problems. *Arvis* try to resolve these problems appropriately and has done quite a lot. As a result, *Arvis* has more than 70 working workflows and this list can continue to be added whenever needed. So, I'm expecting lots of users can work more efficiently with *Arvis*.

References

[1] *Alfred*, Launcher for macOS which boosts efficiency with hotkeys, keywords, text expansion, https://www.alfredapp.com/

[2] packal, Dynamic repository for *Alfred* Workflows and Themes, http://www.packal.org/

[3] sindresorhus, Alfy, https://github.com/sindresorhus/*alfy*

[4] jopemachine, *Arvis*h, https://github.com/jopemachine/*arvish*

[5] jopemachine, *Arvis* linker, https://github.com/jopemachine/*arvis-linker*

[6] jopemachine, *Arvis* notifier, https://github.com/jopemachine/*arvis-notifier*

[7] jopemachine, Generator arvis, https://github.com/jopemachine/*generator-arvis*

[8] jopemachine, *Arvis*h test, https://github.com/jopemachine/*arvish-test*

[9] jopemachine. *Arvis* core, https://github.com/jopemachine/*arvis-core*

[10] jopemachine, Convert alfred 4 workflow's info.plist to *arvis-workflow.json*、 https://github.com/jopemachine/*alfred-to-arvis*

[11] jopemachine, Extension validator for *Arvis* https://github.com/jopemachine/*arvis-extension-validator*

[12] jopemachine, *Arvis* store, https://github.com/jopemachine/*arvis-store*

[13] amilajack, electron-react-boilerplate, https://github.com/electron-react-boilerplate/electron-react-boilerplate

[14] Webpack, https://github.com/*webpack*/*webpack*

[15] Babel, The compiler for writing next generation JavaScript, https://github.com/*babel*/*babel*

[16] wilix-team, *iohook*, https://github.com/wilix-team/*iohook*

[17] Fsevents, https://github.com/fsevents/fsevents

[18] octalmage, robotjs, https://github.com/octalmage/robotjs

[19] pranesh239, use-key-capture, https://github.com/pranesh239/use-key-capture

[20] paulmillr, *chokidar*, https://github.com/paulmillr/*chokidar*

[21] rt2zz, redux-persist, https://github.com/rt2zz/redux-persist

[22] psperber, redux-persist-electron-storage, https://github.com/psperber/redux-persist-electron-storage

[23] Redux, Predictable state container for JavaScript apps, https://github.com/reduxjs/redux

[24] Wox, https://github.com/Wox-launcher/Wox

[25] Dext, https://github.com/DextApp/dext

[26] Cerebro, https://github.com/cerebroapp/cerebro

[27] Zazu, https://github.com/bayleeadamoss/zazu