Table of Contents

List of figures

List of tables

Ⅰ. Introduction

Computer users usually have their own usage pattern. For example, some user may find Chrome history more often than other users. If there is a kind of customized application for finding Chrome history, the user can reduce his/her work time efficiently. In macOS, there is a application named Alfred to make computer users do their work efficient through customized extension called ‘workflow’. Lots of mac user use this program and many workflows for productivity, but Alfred is not free and only runs on mac.

Arvis aims to be cross-platform launcher resolving this problem, based on open source. This document describe about Arvis and related libraries’s implementations. This document does not focus on Arvis’s usage, feature or extension development. (I made other resources about these including documentation homepage)

First, I will describe terminology used in this document, why I'm started making Arvis. Then the reason I chose electron as framework. And will describe about differences with Alfred, and the reasons. Then I will describe related libraries, how they work together, the reason for cloning some libraries, (alfy related). Then I will describe each library, how it works, implementation. And I will describe Arvis's GUI implementations more specifically.

Starting with module structure, process hierarchy used within Arvis, how hotkey feature is implemented, about user config, redux store synchronization with multiple renderer processes, how workflow feature works, how Arvis finds, search commands, how file watcher works.

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 that problems in discussion. At last, I will summarize the paper's content in the conclusion.

II. Background

1) Terminology

Description about some of the terms used in the this paper.

* **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 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 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.

2) Motivation

Lots of alfred-workflows can works in cross-platform but cannot because Alfred only runs in macOS. I always felt like I wanted to use Alfred's workflow when I work on OS other than mac.

It could be a little difficult to find useful alfred workflows. Lots of workflows just exist on github without enough promotion. There is workflow collection site called packal [2], but packal is zombie, haven't been maintained for a long time, isn't familiar with many people, and not all workflows are uploaded in the packal.

It is great to have more advanced quicklook renderer. For instance, alfred-evernote-workflow cache user's notes as form of html, and renders in quicklook window of mac because html is unique option for rendering the contents of note.

I've thought it is very helpful to have more powerful interacting renderer supporting html, pdf, markdown, normal text and more. And I thought if I choose electron loading full feature of chromium, it is enough easy and affordable to support this renderer in cross-platform.

In Alfred, when I use some extension, I must type some command. In some kind of extension, this process might means just waste of time.

For example, when I would like to find some file through extension, I cannot find the file through only typing file name. I must type some command to execute the command.

Finding file through just file name is not attainable through alfred workflow, so

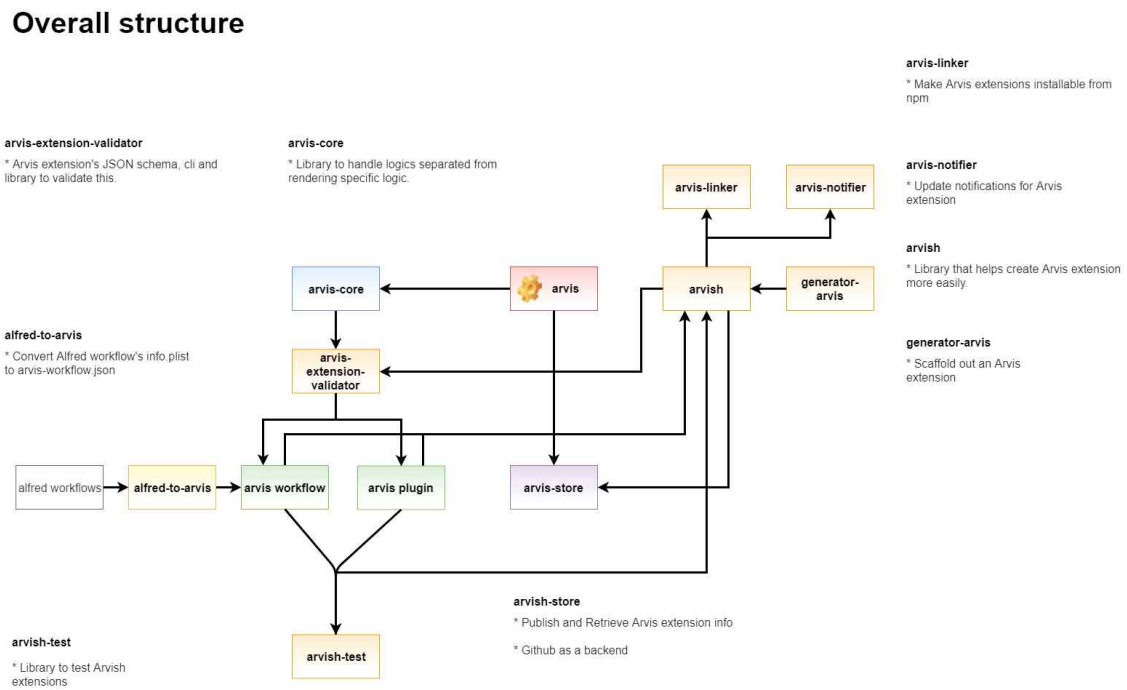
alfred provides some builtin feature like file searching, calculating and other necessary utilities. but I wanted try to make this process other kind of extension.

Arvis supports following features to resolve these problems.

1. Available in 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



<Figure 1> Overall structure

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

2. Related libraries

I decided to clone alfy and related libraries for the following reasons

* I thought unifying programming language with javascript is efficient.

* Lots of alfred-workflow are written in alfy. This means with alfred-to-arvis (converter), I can convert lots of alfred-workflows to arvis-workflows if there is alfy of arvis. So, I decided to alfy and alfy workflows. and I named the cloned library arvish.

* Because I don't want to change alfy API and its behavior, API (on workflow) is basically totally same with alfy. And I thought it is also great to have other libraries in arvish that alfy use. So I cloned the libraries and modify behaviors to work with Arvis. (arvis-linker, arvis-notifier and 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 make Arvis extensions installable from npm. its feature is included in arvish, arvis-linker may be available with other languages than javascript if needed.

3) arvis-notifier

arvis-notifier check extension’s latest version from npm. and if user extension’s version is not latest, mark this on arvis-workflow.json (or arvis-plugin.json). Arvis check each extension’s latest version on arvis-workflow.json on every startup, if there are not latest extension, notify user to update him (her) out-of-dated extensions.

4) generator-arvis

Extension developer can generate skeleton of arvis-workflow.json file, arvis-plugin.json file through arvish. But I think if possible, generating the whole project would be better. generator-arvis is built with yeoman, well-known web scaffolding tool. With generator-arvis, extension developer can scaffold out Arvis extension project's skeleton files easily.

5) arvish-test

arvish-test is test library for arvis-extensions. arvis-extensions needs Arvis environment variables to run. This is provided from Arvis. So, arvis-extension’s javascript file doesn’t run outside of Arvis. arvish-test provides these environment variables instead of Arvis for testing.

6) arvis-core

At first, I was planning making Arvis CLI module (which use Ink as framework). To support CUI and GUI both, I made arvis-core which is detached from renderer logic. But soon I thought it wouldn't be very useful than I expected first. So, I decided to discard CLI module support and focused on Arvis GUI.

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, 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 include library for validating. Arvis use this library when install new extension to filter not valid extensions which can break Arvis's behavior. Extension developers also can use arvis-extension-validator CLI tool for validating for their extension. (This CLI tool is also included in arvish)

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

9) arvis-store

arvis-store works as ‘Github as a backend’, which means all datas used in arvis-store are stored in github, and these data are updated through Github action.

This 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 new extension to arvis-store, extension developer need to register their github authentication token to arvis-store in cli tool. With this token, arvis-store cli tool can make PR (pull request) adding the extension info to ‘internal/static-store.json’. static-store.json stores the extension's static information like name, creator, description, homepage.

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

There are several types that extension developer can upload. These types are specified when extension developer upload their extension. The type indicates 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

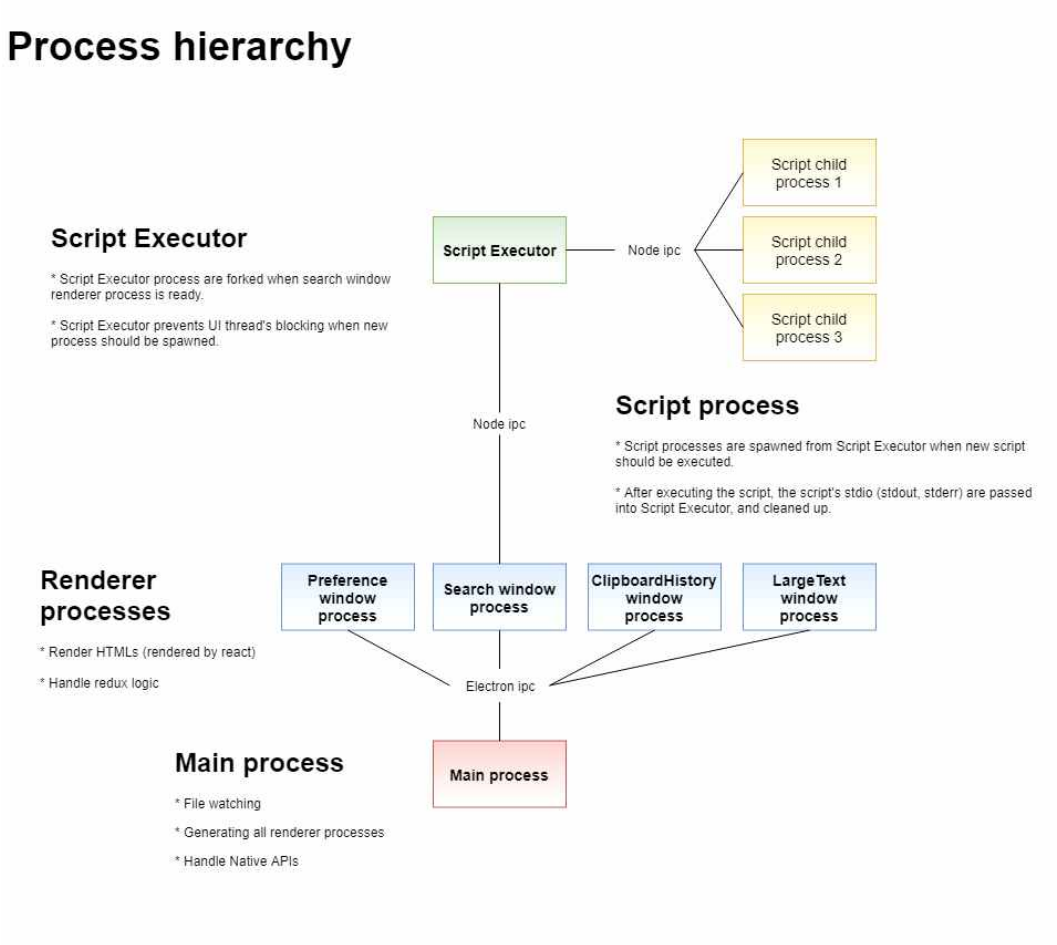
1) Why choose Electron as Framework

Electron supports full feature of chromium. This means lots of feature of chromium are available with minimal efforts. I expected this point to increase the number of works I can do.

Many alfred-workflow are built with alfy (javascript). I've thought I also need to create library help users create their own extension easily. I decided to use clone alfy and I thought to unify programming language with javascript is efficient to work.

Electron has lots of awesome libraries. I thought I could do more works with this awesome libraries than using other platforms in the same time.

2) Process Hierarchy



<Figure 2> Process hierarchy

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 Script Executor runs new script, new child process spawned, previous (other) script child processes are canceled and killed.

III. Arvis GUI Implementation details

1. Module structure

Arvis are built with ‘electron-react-boilerplate’. So, basic project structure and webpack, babel setting files are same with ‘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 in here because they are native modules which means they are builded according to the platform at build time.

‘./package.json’ in the root of your project
: All module except for native modules are specified in this package.json

 # External modules

External modules are ‘src/external’. which are the following.

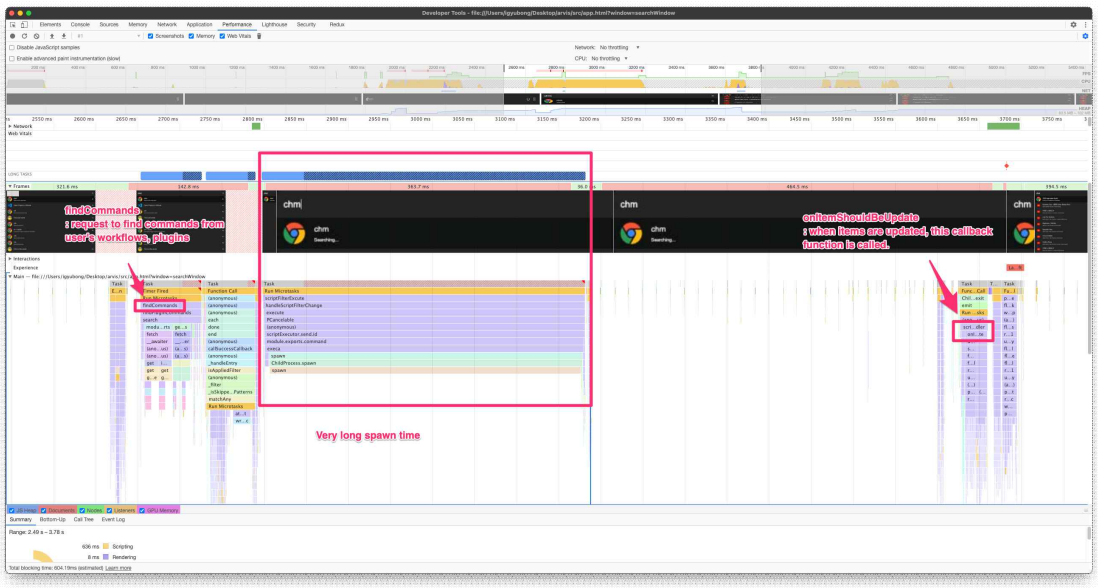
* Type declarations of type-less modules (easy-auto-launch)
* Libraries with extended functionality (use-key-capture)
* Libraries with no webpack-bundling (about-window)
* css files for updating existing module's style
(react-tabs, react-prosidebar, github-markdown-css, jsoneditor)

2. Script Exeuctor

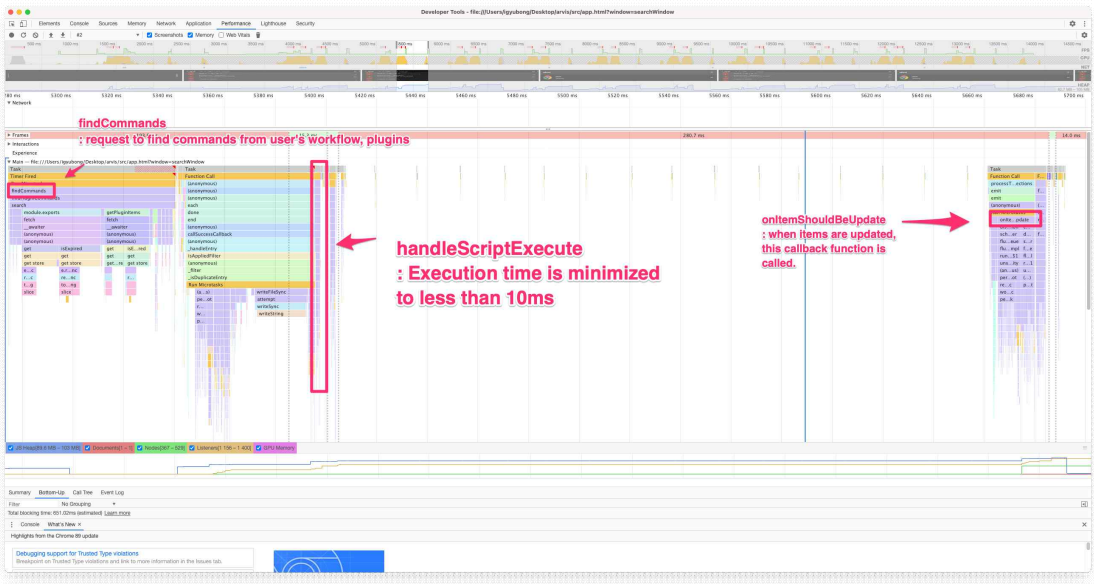
Script Executor process exists for removing process spawning time from search window process when new script should be executed. Without Script Executor process, process spawning time delays renderer process, this leads to the result of main process's UI thread blocking whenever user types something in scriptfilter.

Especially, in mac os Bigsur, this delay time exceeds 150ms which means scriptfilter is barely available to mac users without Script Executor process.

You can check very long spawning time on below image between 'findCommands' function and 'onItemShouldBeUpdate'



<Figure 3> Process spawning without Script Executor process



<Figure 4> Process spawning with Script Executor process

'handleScriptExecute' is function name 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 uses 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 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 keycode, Arvis translate this keycode to hotkey string before recording. For this transformation, Arvis use ‘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 function for translating keycode and hotkey string.

 # Double key handling

Arvis support modifier double key. For 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, some hotkeys are changed.

 # How double hotkeys works

Registering double hotkeys is not supported in electron/globalshortcut, iohook both. So iohook watchs all modifier key press in keydown event handler. and use timers for each modifier keys, recognize double key press, when modifier key is pressed twice in a very short time.

For this, Arvis uses ‘hooks/utils/useDoubleHotkey’. Because iohook should exist in renderer process, double key press handlers exist in renderer process.

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

 # How default hotkey works (Hotkeys only to certain conditions)

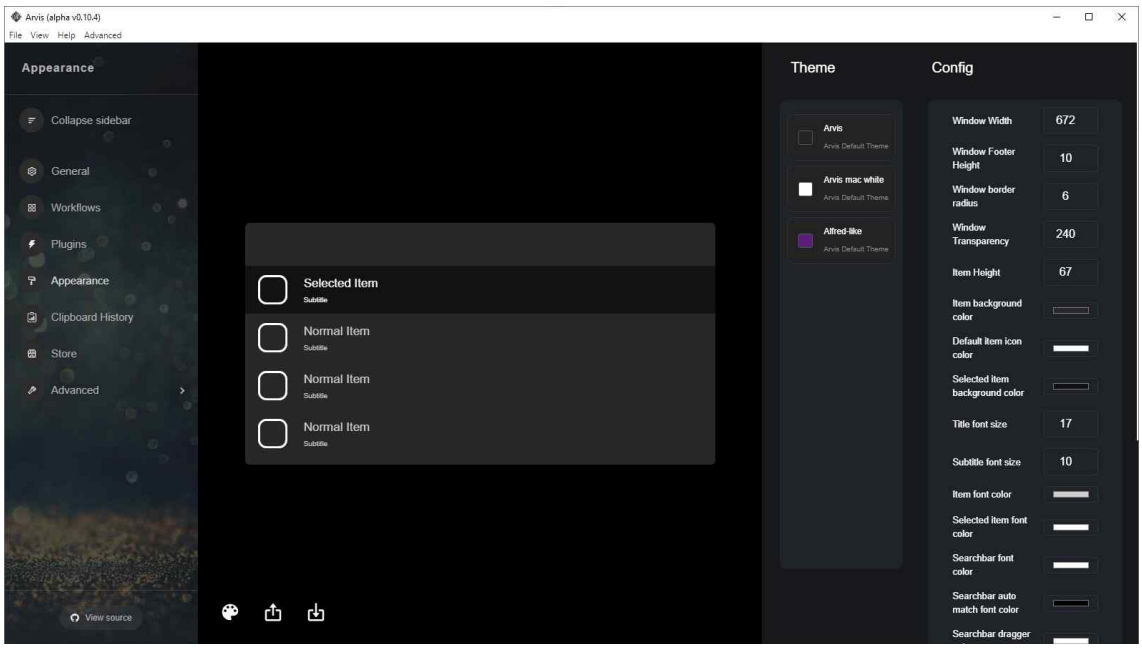
Some hotkeys should be executed only in renderer process. For example, calling large text window only be possible on search window's item. Even if user press the hotkey, large text window should not show up when not on item of search window. So this hotkeys are not registered in separate hotkeys. These hotkeys are handled by use-key-capture's key down event handler.

4. Appearance

Most of variables used in UI styling are stored in redux, and these can be changed by user in Preference window's Appearance page.

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

Users also can export and import their setting values through 'arvistheme' file.



<Figure 5> Appearance page

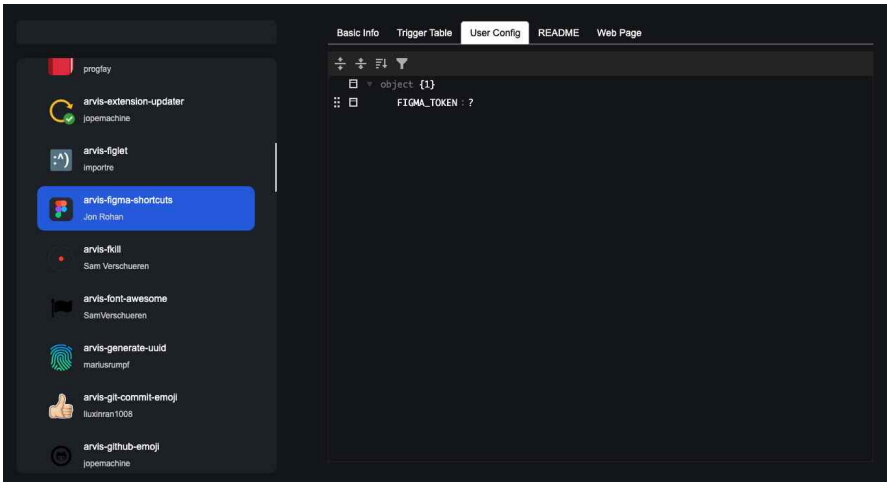
5. User Config

Alfred support setting workflow environment variables, and this variables are stored info.plist. Because those variables are store in info.plist, workflow developers can register necessary variables (like API key) on those tab. Then, users of the workflow can simply put their values on the tab.

Arvis support setting variables on arvis-workflow.json config file. But Arvis's variable values are store in separate file (as user-config file) at the same time. And after the extension update, existing setting values are updated by the user-config file.

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

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



<Figure 6> User config

6. File watcher

Arvis use chokidar to keep watch file change event. File watching is using on extension changes.

For example, if new arvis-workflow.json file is created on extensions folder, Arvis detect it, trigger extension-reload function. By doing this, after installing new extensions, Arvis can reload its extensions.

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

But on plugin extension folder, chokidar watchs 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 consume. 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 file of custom plugin to support hotload, I think reducing system resources is more important than hotload support,

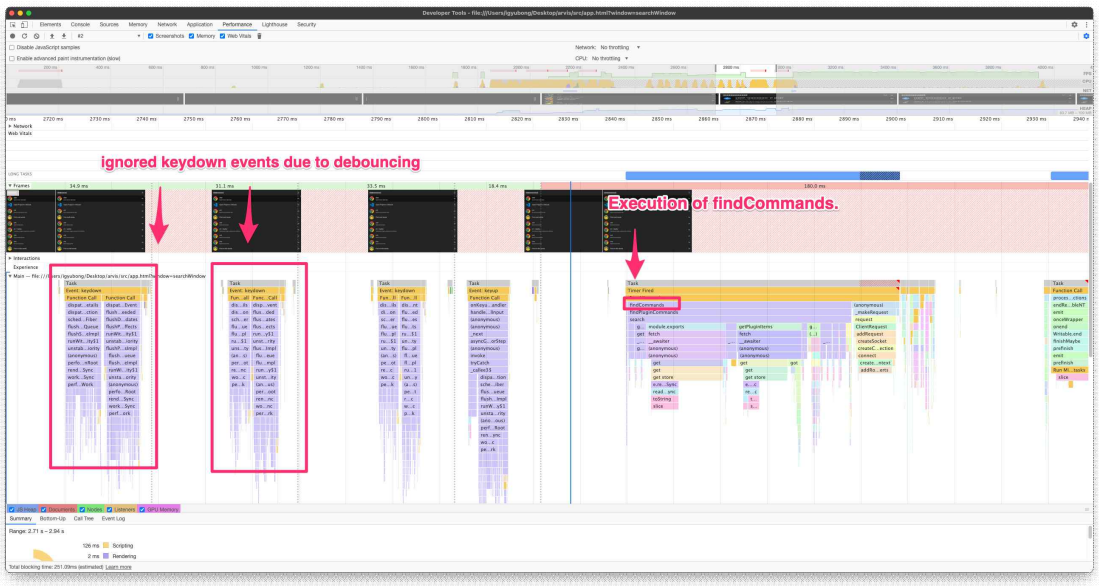
So I restricted monitoring to root folder's javascript files (and arvis-plugin.json file) only.

7. Debouncing optimization

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

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

You can see there is 25ms between last ‘keyup’ Event and ‘findCommand’ function, and other keydown events are all ignored because they are consecutive in 25ms on below figure.



<Figure 7> Optimization through debouncing

8. Global state management

1) The things stored 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

2) How to dispatch action to other renderer processes

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

If any redux-state is changed in some renderer process, the process notify main process redux-state's change through electron ipc module. Then main process send this action to destination renderer process.

3) How to persist redux state

To persist redux state, Arvis use ‘redux-persist’, ‘redux-persist-electron-storage’. This file is stored in 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 current store's states at startup. If there are different points, Arvis are forced to be terminated to reset 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 changes 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 user's custom setting values before restart.

5) Values not stored in redux

Arvis does not store below setting values for following reasons.

* If search window is pinning, clipboard is pinning
: Arvis doesn't consider these states should be saved because Arvis consider the pin state should be off default.

* If preference window's spinner is spinning.
: Arvis does not store dynamically changing UI state (like spinner state) in redux. I think saving these states doesn't make sense because it would be meaningless next time.

* Extension's informations
: Extension's informations could be too big to save one file if the user use a lot of extensions, and Extension's informations is no need to be state. So it's just loaded on memory as form of Record object by arvis-core. Finally, arvis-core's logic should be independent from redux logic.

9. Others

1) Why use arvis-workflow.json instead of info.plist

(info.plist means config file of alfred workflow here.)

* info.plist is not human-readable. info.plist have kind of DAG (directed acyclic graph), this make very difficult to read info.plist. So I decided to make json format replacing the info.plist. I replaced the DAG structure with simple json tree made up of trigger and action, and removed each node's position information.

(The reason I wanted to make config file human-readable is I wasn't planning to make extension GUI creator tool unlike alfred, So I needed to more human-readable config format necessarily)

* info.plist belongs to alfred. it can be updated or changed without notice anytime, I thought using this in Arvis is not reasonable.

IV. Evaluation

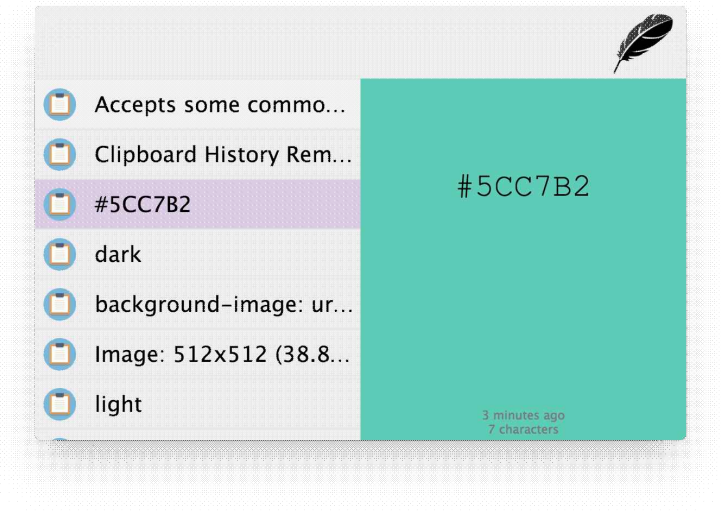
1. Feature comparison

The programs in the ‘table 1’ are all extensible launchers (Based open source other than Alfred)

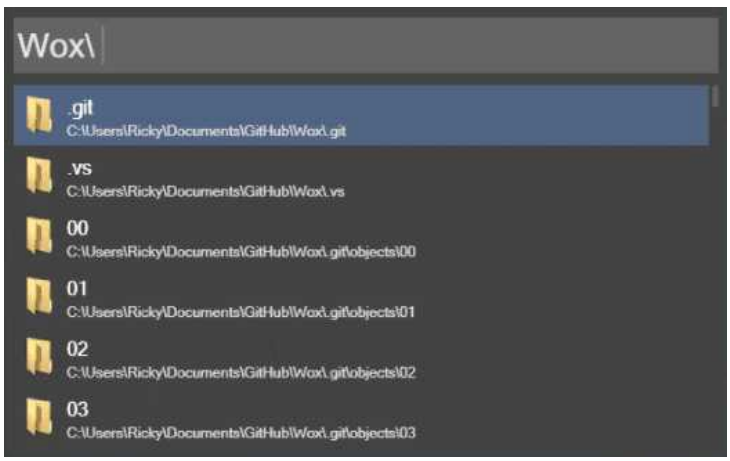
<Table-1> Feature comparison table

	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, mac, linux)	O	X (mac)	X (Windows)	X (mac)	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?	O	O	X	X	X	X

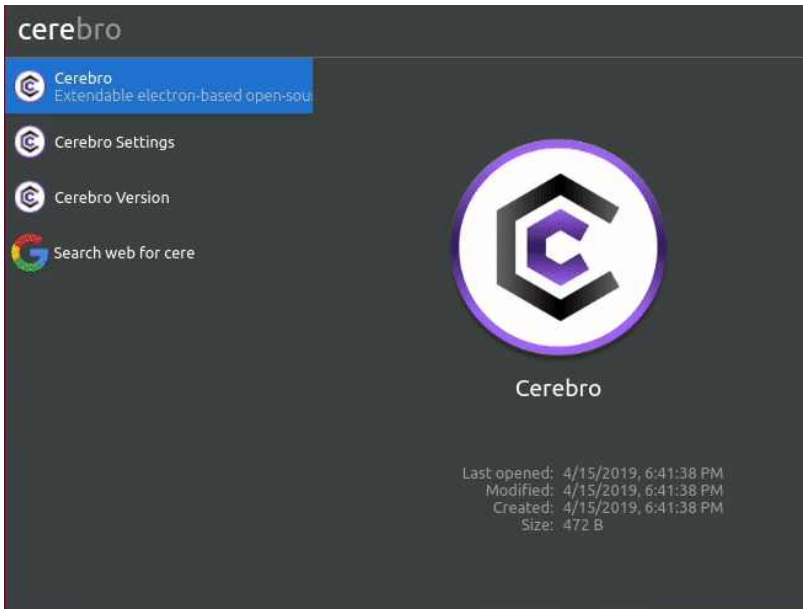
Wox only runs in Windows, Wox doesn't been maintained at the moment. dext have compatibility with alfy workflow, but only runs on mac. Cerebro supports linux, mac, windows. But doesn’t compatible with alfred workflows, and it's been a long time since the last release. Zazu is also been a long time since the last release. And in common, these launchers not support clipboard history, snippet.



<Figure 8> Zazu



<Figure 9> Wox



<Figure 10> Cerebro

<Figure 11> Dext

V. Related works

As described in ‘table 1’, Wox, Dext, Cerebro, Zazu are all kinds of extensible launcher applications. Their difference points are described on ‘Evaluation’.

Compared with ‘automator’ of mac, automator is comprehensive productivity tool to control workflow. But installing, sharing these workflows are not easy. Above all, custom workflows only can be written by Applescript (or JXA), which is too slow and has fewer users compared to other mainstream programming languages. And only runs on macOS.

VI. Discussion

1. Arvis extension auto reload not works with symbolic link directory

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

2. Arvis are freezing at startsup for a while

: I think too heavy operations blocked UI thread currently at startsup. It seems this issue can be resolved through moving this async operation to other process or web worker.

3. Arvis uses too many memory

: Because electron fully loads chromium, electron app inevitably use lots of memory. Additionally Arvis use more memory to render 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 pattern. To boost user's work productivity, users need 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] alfy: sindresorhus, Create Alfred workflows with ease, https://github.com/sindresorhus/alfy

[4] arvish: Lee Gyubong, Arvis workflow and plugin creator tool, https://github.com/jopemachine/arvish

[5] arvis-linker: Lee Gyubong, Make Arvis extensions installable from npm, https://github.com/jopemachine/arvis-linker

[6] arvis-notifier: Lee Gyubong, Update notifications for Arvis extension、 https://github.com/jopemachine/arvis-notifier

[7] generator-arvis: Lee Gyubong, Scaffold out an Arvis extension、 https://github.com/jopemachine/generator-arvis

[8] arvish-test: Lee Gyubong, Test your Arvish extensions、 https://github.com/jopemachine/arvish-test

[9] arvis-core: Lee Gyubong, Arvis core module, https://github.com/jopemachine/arvis-core

[10] alfred-to-arvis: Lee Gyubong, Convert alfred 4 workflow's info.plist to arvis-workflow.jso n、https://github.com/jopemachine/alfred-to-arvis

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

[12] arvis-store: Lee Gyubong, Arvis store module, https://github.com/jopemachine/arvis-store

[13] electron-react-boilerplate: amilajack, A Foundation for Scalable Cross-Platform Apps, https://github.com/electron-react-boilerplate/electron-react-boilerplate

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

[15] babel, Babel is a tool that helps you write code in the latest version of JavaScript, https://github.com/babel/babel

[16] iohook: wilix-team, Node.js global keyboard and mouse listener. https://github.com/wilix-team/iohook

[17] fsevents: Native access to MacOS FSEvents in Node.js, https://github.com/fsevents/fsevents

[18] robotjs, octaImage, Node.js desktop automation, https://github.com/octalmage/robotjs

[19] use-key-capture: pranesh239, Makes listening to key press event easy., https://github.com/pranesh239/use-key-capture
[20] chokidar, Minimal and efficient cross-platform file watching library, https://github.com/paulmillr/chokidar
[21] redux-persist, persist and rehydrate a redux store, https://github.com/rt2zz/redux-persist
[22] redux-persist-electron-storage, Redux persist adapter for electron-store, https://github.com/psperber/redux-persist-electron-storage
[23] redux, Predictable state container for JavaScript apps, https://github.com/reduxjs/redux
[24] wox, Launcher for Windows, an alternative to Alfred and Launchy, https://github.com/Wox-launcher/Wox
[25] dext, A smart launcher. Powered by JavaScript., https://github.com/DextApp/dext
[26] cerebro, Open-source productivity booster with a brain, https://github.com/cerebroapp/cerebro
[27] zazu, A fully extensible and open source launcher for hackers, creators and dabblers, https://github.com/bayleeadamoss/zazu