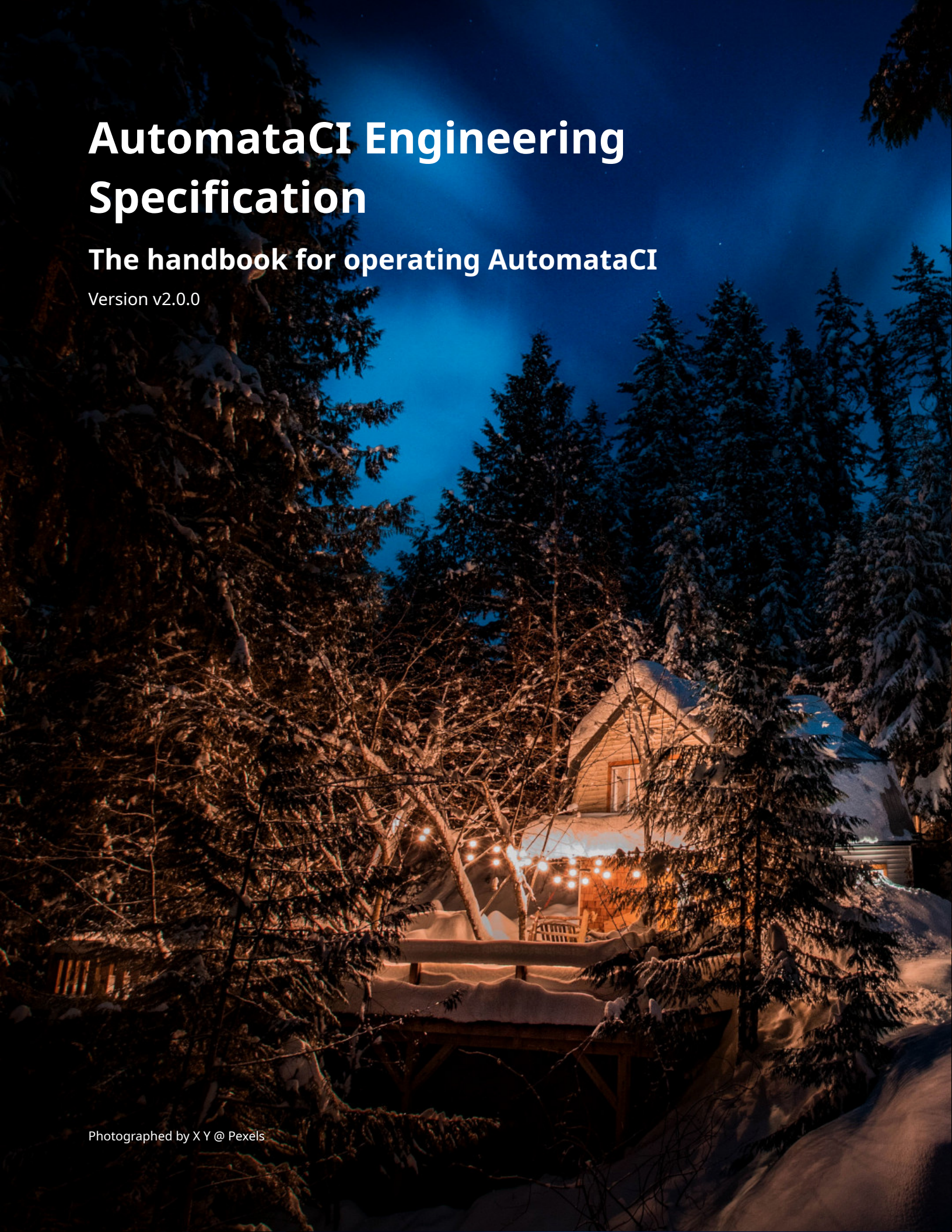
AutomataCI Engineering Specification

The handbook for operating AutomataCI

Version v2.0.0



Photographed by X Y @ Pexels

1. About the Book

AutomataCI Engineering Specification - *The handbook for operating AutomataCI*

Version v2.0.0

Authored by:

(Holloway) Chew, Kean Ho
Independent
hollowaykeanho@gmail.com

Published by:

AutomataCI & GitHub
https://github.com/ChewKeanHo/AutomataCI

To Reference This Material:
Chew, Kean Ho; 2023; "AutomataCI Engineering Specification"; *AutomataCI Software Repository*; Version v2.0.0; DOI: DOI_HERE; AutomataCI Engineering Team via GitHub.com; Accessed on <DATE>; Available at: https://github.com/ChewKeanHo/AutomataCI

2. Abstract

Continuous improvements or integration (CI) is a nerve system of a software product production. Leaving the job blindly by outsourcing to any proprietary 3rd-party CI service provider is a guaranteed way for being threatened and extorted by any malicious suppliers. As of year 2021 to 2023, many evidences show that the suppliers vilely alter their pricing charges after the fact to the point of legally extorting their customers and destroyed some useful software entirely.

Therefore, AutomataCI counters such problem by facilitating CI functions natively, making it locally available, decentralized, and redistribution at git repository level. 3rd-party service providers can only call AutomataCI just like how the developer performs them locally. This de-risk the software owner by not completely depending on the service provider, allowing him/her to switch supplier easily. AutomataCI also facilitates technology-specific automation customization, capable of absorbing new changes easily across time. It only uses the basic POSIX compliant Shell and PowerShell scripts.

This book covers the user manual and the engineering specifications required to further develop or to maintain the current AutomataCI source codes. The goal is to ensure the knowledge is always available even in the disconnected conditions.

3. Table of Contents

4. Prologue

Thank you for selecting and using AutomataCI for your project. While you're having a solid business reason to deploy AutomataCI, this specification ensures the knowledge and freedom of owning your copy of AutomataCI is preserved alongside your repository in the event of upstream failure (where the origin of AutomataCI is lost for any reason).

Should you be interested to contact my team and I at the upstream level, please feel free to visit the following portals for:

1 Chit chat or casual discussion: https://github.com/orgs/ChewKeanHo/discussions

2 AutomataCI upstream repository: https://github.com/ChewKeanHo/AutomataCI

4.1. Why Another CI

AutomataCI was specifically built to counter supply-chain threat encountered since Year 2022 across the Internet service providers. A white paper is available for detailing of the incident and for case study education purposes is available. (refer: https://doi.org/10.5281/zenodo.6815013). Ever-since the post Covid-19 pandemic, a lot of CI service providers are drastically changing their business offering to the extent of extorting their customers to either pay a very high price or close the entire project operation down.

In response to such threat, ZORALab's Monteur (https://github.com/zoralab/monteur) was first created to remove such a threat. However, it has its own flaws dealing with various OSes native functions. Hence, AutomataCI was iteratively created to resolve ZORALab's Monteur weakness, allowing a project repository to operate without depending entirely on ZORALab's Monteur's executable.

4.2. The AutomataCI Mantra

The sole reason for deploying a CI from the get-go is to make sure the project life-cycle can be carried out consistently anywhere and anytime. The project is designed for both remote executions in the cloud and even locally. It facilitates heavier resistances and resilience to market changes without hampering the product development and production progress. Unlike other CI models, ***AutomataCI favors the "semi-automatic" approach*** where the automation can also be manually executed stage-by-stage or fully automated, at will.

This provides decent protection against ill-intent vendors from legally extorting or ransoming you via vendor locked-in and after-the-fact business changes. If a CI service provider turns the relationship sour, one can easily switch to another. Another good reason is the decoupling effect done to the CI, allowing developers to specifically test a pipeline job manually and locally.

4.2.1. Cautiously Integrating External Service Providers

To counter the external CI service providers' threatening changes, **their service' interfaces is strictly ONLY to call AutomataCI job executions just like a regular developer does in the local machine**. Therefore, most of the time, you will be using AutomataCI's service libraries in both of your POSIX Shell and PowerShell scripting instead. By deploying adapter approach, it's easier to maintain and isolate all 3rd-party service providers without hampering the customers' project or AutomataCI maintenance teams.

Due to the fact that CI is an important life-support system for your project, **you're strongly advised not to use any vendor locked-in API or functionalities**. Anything AutomataCI can't do locally signaling that it is vendor locked-in solution. The more such you use, the more entangled you are; which also means the more painful for you to do immediate migration when threat suddenly appear.

5. Quick Start

This section covers a quick re-cap on how to operate AutomataCI and get to work. This is arranged here at the get-go primarily for newcomers and developers who are unfamiliar with with the supported repository. Regardless, if you're new, please go through this specification at full at least 1 time and get to know better with the infrastructure.

The sub-sections are steps which are arranged in a storytelling sequences.

5.1. Downloading the Repository

You need to contact your repository maintainer for downloading a copy of your project repository. Example, for those that uses git version control system, the command is usually:

```
$ git clone <PROJECT_URL>
$ cd <YOUR_PROJECT>
```

5.2. CONFIG.toml Defines the Project and Repository

Please go through the *CONFIG.toml* file located at the root directory of the repository. This configuration file defines the project entirely.

IMPORTANT REMINDER

The TOML file can only and strictly accept "key = value" format. Please do not get too fancy.

5.3. [OPTIONAL] Provide Secret Variables or SECRETS.toml

Depending on the project and external services interfacing, chances are there will be secrets data to deal with. You can either supply each of them via the environment variable interface or request a *SECRETS.toml* configuration file from your maintainer. This file extends the *CONFIG.toml* file where the data are not supposedly to be committed in the version control system and share the exact properties with *CONFIG.toml* file.

There is a template file for education and reference purposes located in *automataCI/* directory (called *SECRETS-template.toml*). If you're creating one from fresh, please feel free to duplicate that template file.

5.4. To Check and Setup Host Machine Environment

To quick configure a brand new computing machine hosting the AutomataCI run (called "host machine"), you can simply run the following command:

```
$ ./automataCI/ci.sh.ps1 env
```

Upon completion, your host machine should contain the vital tools for AutomataCI to run.

5.5. To Setup The Repository For Development & Automation

To setup the repository for your project development and automation run (example: the project's programming languages, etc), you can simply run the following command:

```
$ ./automataCI/ci.sh.ps1 setup
```

Upon completion, your repository is ready for your project development.

5.6. To Prepare The Project

To prepare the project repository with all the dependencies pulled-in and be ready for next steps, depending on technologies, you can simply run the following command and follow through its instructions:

```
$ ./automataCI/ci.sh.ps1 prepare
```

Upon completion, your repository is ready for your project development.

5.7. To Start A Development

To start a development, depending on technologies which may contain some virtual environment initialization, you can simply run the following command and follow through its instructions:

```
$ ./automataCI/ci.sh.ps1 start
```

Upon completion, your repository is ready for your project development.

5.8. To Test The Project

To perform a full test cycle for the entire project, you can simply run the following command:

```
$ ./automataCI/ci.sh.ps1 test
```

Upon completion, depending on your maintainers' configurations and availability, you should be able to find the test report in the *${PROJECT_PATH_ROOT}/${PROJECT_PATH_LOG}/* directory.

5.9. To Materialize Project for Host Machine

To build the product only for host machine's use, you can simply run the following command:

```
$ ./automataCI/.ci.sh.ps1 materialize
```

Upon completion, you should find:

1 the executables are in *${PROJECT_PATH_ROOT}/${PROJECT_PATH_BIN}/* directory

2 the libraries are in *${PROJECT_PATH_ROOT}/${PROJECT_PATH_LIB}/* directory.

5.10. To Build The Products

To build all the products, you can simply run the following command:

```
$ ./automataCI/.ci.sh.ps1 build
```

Upon completion, you should find:

1 the built artifacts in *${PROJECT_PATH_ROOT}/${PROJECT_PATH_BUILD}/* directory.

5.11. To Notarize All Built Artifacts

To notarize all the built artifacts, you can simply run the following command:

```
$ ./automataCI/.ci.sh.ps1 notarize
```

Upon completion, you should find some compatible built artifact are signed.

GENTLE REMINDER

It's your duty to have the required certificates and secrets in-place before executing this CI job run.

5.12. To Package The Products

To package the built artifacts and the project into consumer products, you can simply run the following command:

```
$ ./automataCI/ci.sh.ps1 package
```

Upon completion, you should find:

1 the built artifacts in *${PROJECT_PATH_ROOT}/${PROJECT_PATH_PKG}/* directory.

Upon completion, you should find some compatible built artifact are signed.

GENTLE REMINDER

It’s your duty to have the required certificates and secrets in-place before executing this CI job run.

5.13. To Certify The Packages

To notarize all the packages, you can simply run the following command:

```
$ ./automataCI/ci.sh.ps1 certify
```

Upon completion, you should find some compatible packages are signed.

GENTLE REMINDER

It’s your duty to have the required certificates and secrets in-place before executing this CI job run.

5.14. To Release The Packages

To release the packages, you can simply run the following command:

```
$ ./automataCI/ci.sh.ps1 release
```

Upon completion, you can check your published store for the released packages.

5.15. To Deploy The New Release

To deploy the new release in your server system when applicable, you can simply run the following command:

```
$ ./automataCI/ci.sh.ps1 deploy
```

Upon completion, you can check your ecosystem for the updated deployment.

5.16. To Clean the Project

To clean up the project for the next build, you can simply run the following command:

```
$ ./automataCI/ci.sh.ps1 clean
```

Upon completion, depending on availability, the project should be cleansed of artifacts for the next job.

5.17. To Purge the Project

To reset the project to its initial condition (except host machine environment setup), you can simply run the following command:

```
$ ./automataCI/ci.sh.ps1 purge
```

Upon completion, your project repository is reset where you need to run setup job from scratch again.

5.18. To Customize the CI Job

To customize one or more CI job recipes, you can modify them in the source directory. Example:

1. **Baseline (affects all) –** *${PROJECT_PATH_SOURCE}/.ci/*
2. **Go** – *${PROJECT_GO/.ci/*
3. **Rust** – *${PROJECT_RUST}/.ci/*
4. **Angular** – *${PROJECT_ANGULAR}/.ci/*
5. ...

IMPORTANT REMINDER

DO NOT edit anything inside the *automataCI/* directory. All changes within shall be brutally overridden should there be updated (see next sub-section).

5.19. To Upgrade AutomataCI

To upgrade the AutomataCI, there is a checklist to comply with:

1. **Determine the business need** – check the AutomataCI release notes and make sure there is something you need in the later version. Otherwise, you really need to upgrade towards latest and greatest at all.
2. **Overwrite the *automataCI/* directory** – Download the later version of AutomataCI and only overwrites the local *automataCI/* directory.
3. **Update *CONFIG.toml*** – Cross-check the later version's changes and patch it.
4. **Apply additional changes if instructed** – Should the release note instructs additional changes, please cross-check and apply them.
5. **Update the common source's CI job recipes** – First update the common source CI job recipe accordingly.
6. **Update the project's tech-specific CI job recipes** – Update all the project's tech specific CI job recipes based on the later version's features and changes if needed.
7. **Test and re-run** – Test and re-run entirely. Repeat from Step 2 if required.
8. **Commit the changes** – Should the pipeline is working fine. You can commit and AutomataCI is updated.

5.20. To Integrate AutomataCI Into Existing Project

To integrate the AutomataCI into existing project, due to the intense customization nature of a project being managed from one to another, AutomataCI offers a checklist to ensure consistency while retaining the project's customization capability:

1. **Evaluate current project's commands executions** – make sure you write down all the required commands to operate your existing project. Please keep them in a source directory (e.g. *srcGO/* for Go project, etc) and have the root directory clean.
2. **Download and install *automataCI/* directory** – perform the AutomataCI installation first.
3. **Install *CONFIG.toml*** – place the configuration file at the root of the repository and modify its values accordingly.
4. **Setup the project path in *CONFIG.toml*** – depending on how you arrange your directory in step 1, you need to configure the project path matching accordingly to the technology (e.g. PROJECT_GO = "*srcGO/*"). If you arrange everything in the baseline directory (as in PROJECT_PATH_SOURCE = "*src/*"), then skip this step.
5. **Install the baseline directory (default is "*src/*")** – AutomataCI requires certain files to be in the baseline directory to operate properly. Please install it accordingly. Should you need to change the name (e.g. something other than "*src/*"), please make sure you update both *$PROJECT_PATH_SOURCE* and *$PROJECT_PATH_RESOURCE* values pointing to the custom directory in the CONFIG.toml file.
6. **Download and install the tech-specific CI job recipes** – If you're using tech-specific source directory (e.g. *srcGO/* for Go project, etc), download and install its CI job recipes and place it inside as *.ci/* directory.
7. **Update the CI job recipes** – At this point, you can update the commands recorded in Step 1 into the CI job recipes (the *.ci/* directory) first from the tech-specific directory (e.g. *srcGO/* for Go project) and then from the baseline directory (default is "*src/*"). See the "CI Job" section for detailed specifications.
8. **Test run and debug** – Perform a test run and debug the problem accordingly.
9. **Add everything and commit the changes** – Should everything are working expected, you should commit everything and push out to your upstream. Don't worry, you can optimize or translate to another counterparts (e.g. POSIX Shell to PowerShell) later.

5.21. To Install AutomataCI as a New Project

To install AutomataCI as a new project, you can:

1. **Download and install *automataCI/* directory** – perform the AutomataCI installation first.
2. **Install *CONFIG.toml*** – place the configuration file at the root of the repository and modify its values accordingly.
3. **Install the baseline directory (default is "*src/*")** – AutomataCI requires certain files to be in the baseline directory to operate properly. Please install it accordingly. Should you need to change the name (e.g. something other than *src/*), please make sure you update both *$PROJECT_PATH_SOURCE* and *$PROJECT_PATH_RESOURCE* values pointing to the custom directory in the *CONFIG.toml* file.
4. **Download and install the tech-specific CI job recipes** – If you're using tech-specific source directory (e.g. "*srcGO/*" for Go project, etc), download and install its CI job recipes and place it inside as *.ci/* directory.
5. **Setup the project path in *CONFIG.toml*** – Once everything is in place, update this configuration file matching your project details.
6. **Test run and debug** – Perform a test run and debug the problem accordingly.
7. **Add everything and commit the changes** – Should everything are working expected, you should commit everything and push out to your upstream. Don't worry, you can optimize or translate to another counterparts (e.g. POSIX Shell to PowerShell) later.

5.21.1. Install Using GitHub

Alternatively, if you have access to GitHub.com (as in, network availability, geo-political embargo restrictions, politician nuisances, or etc), you can create a repository via clicking the "Use this template" button at https://github.com/ChewKeanHo/AutomataCI.

6. Internationalization

This section covers the specifications AutomataCI native communication languages (as in we, human communicates) and how the team came to a rationale conclusion for such deployment.

6.1. Default to International English (en)

AutomataCI fundamentally is a product of Science, Mathematics, Engineering, and Technologies (STEM). Hence, **the default language is always in International English** (code: ***en*** without any country code). As STEM is mainly conversed in English across the Earth, AutomataCI gradually comply to its default. Here are some of the supporting cases:

1. https://old.iupac.org/publications/ci/2013/3503/bw2_moreau.html
2. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3341706/

AutomataCI believes in building windows and doors welcoming people from different background; not walls that discriminates people or becoming a politicians' excuses.

6.2. UTF-8 Encoding

Although AutomataCI defaults to English, by default, AutomataCI shall and always configure itsefl to UTF-8 encoding and using font that supports many other languages such as *Noto Sans* and *Noto Serif* from the Noto Project (https://fonts.google.com/noto).

6.3. Derivative Translation

Depending on the content nature, AutomataCI performs derivative translation from the main International English copy into other languages (when translation is available). This means 2 things:

1. International English version is the most updated and first to be finalized.
2. Should there be any incomplete translation, it is vital to fall back to the default language.

6.4. ISO639-1+ISO-15954+ISO-31661Alpha2 Language Code

AutomataCI uses 3 sets of standards to identify a language for a given location in order to facilitate its services properly especially in the location where geographical fencing is required. While AutomataCI is aware there are other ID systems, this is the preferred one as they're intuitive to be accurate without much guessing. The format is listed in Table 1.

Table 1: AutomataCI Language Code Format

{iso639-1}[-{iso15954}][-iso31661] **{language_code}[-language_variant][-country_code]**
NOTE: 1. All must be in lowercase. 2. Only dash (-) is used for separation. LEGEND: 1. **{ ... }** - Compulsory Element 2. **[...]** - Optional Element

Examples:

1. **en** – International English.
2. **en-us** – English (United States).
3. **fr** – International French.
4. **fr-ca** – Canadian French.
5. **zh-hans** – International Simplified Chinese.
6. **zh-hant** – International Traditional Chinese.
7. **zh-hans-tw** – Taiwanese Simplified Chinese.
8. **zh-hans-hk** – Hong Kong SAR Simplified Chinese.
9. **zh-hans-cn** – Mainland China Simplified Chinese.
10. **zh-hant-tw** – Taiwanese Traditional Chinese.
11. **zh-hant-hk** – Hong Kong SAR Traditional Chinese.
12. **zh-hant-cn** – Mainland China Traditional Chinese.
13. **zh-hans-us** – United States Simplified Chinese.
14. **zh-hant-us** – United States Traditional Chinese.

References:

1. https://developers.google.com/search/docs/specialty/international/localized-versions
2. https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
3. https://unicode.org/iso15924/iso15924-codes.html
4. https://en.wikipedia.org/wiki/ISO_3166-1_alpha-2

6.5. S.I Unit Measurement Compliance

For any measurements, **AutomataCI strictly uses and comply to the International System of Units (S.I) units**. This is for aligning to the international science communities and remove unnecessary conversions when used anywhere in the world.

7. Filesystem

This section covers the filesystem used by AutomataCI as a component in a given file-based repository. The purpose is for checking out any vital files owned by AutomataCI that can make or break its operations.

7.1. File Extensions

AutomataCI complies to the UNIX OSes and Windows OS friendly file extensions such that a compound format is deployed (E.g. ***.tar.gz***).

While UNIX OSes do not use it for software service identification, Windows OS does. Hence, in AutomataCI, the last segment of the file extensions shall always be the first service identifier across all OSes. Some notable examples are:

1. **.sh.ps1.cmd** | **.sh.cmd.ps1** – for POSIX+Batch+PowerShell Polygot Script.
2. **.sh** – for all POSIX shell scripts only.
3. **.ps1** – for all PowerShell scripts only.
4. **.toml** – for CONFIG.toml file only.
5. **.exe** – for Windows exectuables.
6. **lib[NAME].a** – for UNIX C/C++ libraries.
7. **[NAME].dll** – for Windows C/C++ libraries.

7.2. Naming Convention

AutomataCI deploys its own naming convention for maximum consistencies and content identification without needing to perform byte-reading. The said convention is also for manual human comprehension as well. AutomataCI uses the pattern defined in Table 2.

Table 2: the AutomataCI filesystem naming convention format.

{NAME}[-TYPE][_LANG]_{OS}-{ARCH}.{EXTENSION}[/]
NOTES: 1. **underscore (_)** is for context switching. 2. **dash (-)** for separating different subjects within the same context. 3. **space ( ) is prohibited** (NOT ALLOWED). 4. For directory type, it is always having **the backslash suffix (/)**. LEGENDS: 1. **{ ... }** - compulsory field. 2. **[...]** - optional field.

Examples:

1. **setup_windows-amd64.ps1** – PowerShell script in default language running setup function for Windows OS, with *amd64* CPU only.
2. **setup_windows-any.ps1** – PowerShell script in default language running setup function for Windows OS with any CPU types.
3. **setup_unix-amd64.sh** – POSIX shell script in default language running setup function for UNIX OS (Linux, Hurd, and Apple MacOS) with *amd64* CPU only.
4. **setup_unix-any.sh** – POSIX shell script in default language running setup function for UNIX OS with any CPU types.
5. **setup_darwin-any.sh** – POSIX shell script in default language running setup function for Apple MacOS only with any CPU types.
6. **myapp-src_unix-any/** - Directory containing the project “myapp” source codes in default language deploy-able for UNIX OS with any CPU types only.

7. **myapp-src_en_unix-any/** - Directory containing the project “myapp” source codes in International English deploy-able for UNIX OS with any CPU types only.
8. **myapp_windows-amd64.exe** – An executable called “myapp” in default language with *.exe* file extension deploy-able only for Windows OS with *amd64* CPU.
9. **myapp_zh-hans_windows-amd64.exe** – An executable called “myapp” in International Simplified Chinese language with *.exe* file extension deploy-able for Windows OS with *amd64* CPU only.
10. **LICENSE_en.rtf** – a rich text document in International English usable anywhere.

7.3. Repository File Structures

Within an AutomataCI integrated Repository File, depending on the definitions in *CONFIG.toml*, by default, the affected files and directories are listed in Table 3.

Table 3: The AutomataCI default filesystem

automataci/	house the projects' CI automation scripts.
automataCI/ci.sh.ps1	CI start point sourcing *automataCI/ci.sh* or *automataCI/ci.ps1*.
automataCI/ci.sh	AutomataCI entrance for UNIX OS.
automataCI/ci.ps1	AutomataCI entrance for Windows OS.
automataCI/services/	house tested and pre-built CI automation services' functions.
bin/	default removable materialized output directory for executables
build/	default removable build output directory.
lib/	default removable materialized output directory for libraries.
pkg/	default removable package output directory.
src/	house common assets and materials (baseline directory).
src/packages/	housing all packages control template files.
src/icons/	housing all graphics and icon files.
src/icons/icons.ico	artifact icon in .ICO format used in Windows OS.
src/icons/msi-banner-top_[LANG].jpg	artifact banner used in MSI packaging.
src/icons/msi-dialog_[LANG].jpg	artifact banner used in MSI packaging.
src/licenses/	housing all project licensing generative documents and artifacts.
src/licenses/deb-copyright	extended license file used in DEB packaging.
src/licenses/LICENSE_[LANG].rtf	license file artifact used in MSI packaging EULA content display.
src/licenses/LICENSE_[LANG].pdf	license file artifact used for general packaging distribution.
src/docs/	housing all project's document generators

src/changelog/	housing all project’s changelog entries data.
src/.ci/	house all baseline (applies to all) common CI job recipes.
src[TECH]/	house tech-specific source codes (tech-specific directory).
src[TECH]/.ci/	house tech-specific CI job recipes.
tools/	default tooling (e.g. programming language) directory.
tmp/	default removable temporary workspace directory.
CITATION.cff	default auto-generated academic citation data file.
CONFIG.toml	configure project's settings data for AutomataCI.
SECRETS.toml	provides secrets related data (ignored by default .gitignore).
.git	default Git version control configuration directory.

7.4. Secret Files

AutomataCI does facilitate secret files parsing for sourcing confidential data such as API token and etc called **SECRETS.toml**. This file is usually located next to the *CONFIG.toml* at the root directory of the repository.

When using the default settings, the ***.gitignore*** shall always ignore such file to prevent any data leak.

Alternatively, you're free to provide those secret parameters either through this file or define those environment variables directly on your own.

8. Platform & OS Definitions

This section covers the system platform and operating system (OS) definitions used by AutomataCI as the system-level identification purposes.

8.1. System Definition Format

The definition complies to the format tabulated in Table 4. By default, AutomataCI uses the lowercase uname list (see: https://en.wikipedia.org/wiki/Uname#Examples) output as its sourcing data. However, some special or exotic values are inexplicably converted to a common value (detailed in next sub-section).

Table 4: AutomataCI Platform Definitions

{platform} = {os}-{arch}
NOTES: 1. All shall be in lowercase. 2. The above can be interpreted interchangeably. LEGENDS: 1. **{ ... }** - compulsory field. 2. **[...]** - optional field.

Example:

1. **linux-amd64** – Linux OS + AMD64 CPU only
2. **linux-any** – Linus OS only
3. **any-any** – Any OS & Any CPU
4. **darwin-amd64** – Mac OSX with Intel CPU
5. **darwin-arm64** – Mac OSX with Apple M-Series CPU
6. **windows-amd64** – Windows + AMD64 CPU only

8.2. Special Cases Re-Definitions

It's duly noted that some computing platforms available in the market are too exotic and unique. For those cases, AutomataCI shall re-define them back to their common settings as AutomataCI do not intend to support them directly due to market-locked in nature. They're tabulated in Table 5 and Table 6.

Table 5: AutomataCI redefined CPU Architecture Values

ARCH	**Description**	**Denoted As**
i686-64	Intel Itanium CPU	*ia64*
i386, i486, i586, i686	Intel X86 32-bit CPU	*i386*
x86_64	Intel/AMD X86 64-bit CPU	*amd64*
sun4u	TI Ultrasparc	*sparc*
power macintosh	IBM PowerPC	*powerpc*
*ip**	MIPS CPU	*mips*

Table 6: AutomataCI redefined OS Values

OS	**Description**	**Denoted As**
windows, ms-dos**	Microsoft Windows OSes	*windows*
cygwin, mingw*, mingw32*, msys**	GCC/Linux emulator in Windows OSes	*windows*
**freebsd*	FreeBSD OSes	*freebsd*
*dragonfly**	Dragonfly OSes	*dragonfly*
standalone, unknown, none	Bare-metal/microcontroller, binary image/fat binary, unikernel image	*none*

8.3. Windows CPU Architectures' Value Interpreters

Due to Microsoft using its own values for CPU architectures (see: https://learn.microsoft.com/en-us/dotnet/api/microsoft.powershell.commands.cpuarchitecture?view=powershellsdk-1.1.0), AutomataCI made 2 intepreter functions in the Microsoft publisher library suite and only use it near the output generative function. Internally, AutomataCI does not use them even in its Windows PowerShell scripts. Table 7 demonstrates the code snippets for how to use them interchangeably.

Table 7: the AutomataCI filesystem naming convention format.

```
# POSIX Shell
. "${LIBS_AUTOMATACI}/services/publishers/microsoft.sh"

__arch="amd64"

__arch="$(MICROSOFT_Arch_Interpret "$__arch")"
1>&2 printf -- "%s" "$__arch" # which is "x64"

__arch="$(MICROSOFT_Arch_Get "$__arch")"
1>&2 printf -- "%s" "$__arch" # which is "amd64"
```

```
# PowerShell
. "${env:LIBS_AUTOMATACI}/services/publishers/ microsoft.ps1"

$__arch = "amd64"

$__arch = MICROSOFT-Arch-Interpret "$__arch"
Write-Host "Got: ${__arch}" # which is "x64"

$__arch = MICROSOFT-Arch-Get "$__arch"
Write-Host "Got: ${__arch}" # which is "amd64"
```

9. CI Jobs

This section covers the definition of all the CI jobs offered by AutomataCI. They arranged and tested as a linear storyline for keeping the process sane and simple to comprehend.

9.1. General Governance

AutomataCI employs a linear story-line of factory production where one can visualize how a product is produced from raw materials into manufactured product, packaging, distribute, and lastly consume the product as the end user. That way, it's easy to cope and map AutomataCI applications to the real-world processes.

9.1.1. Consistent Yet Customizable

While AutomataCI maintains high consistency in its software manufacturing processes, it still facilitate maintainable customization capabilities via the source's *.ci/* directory that houses specialized and custom CI implementation recipes. For example,

1. For Python technology, the *${PROJECT_PATH_ROOT}/${PROJECT_PYTHON}/.ci/setup_unix-any.sh* houses only the setup implementations for Python technology itself.
2. For Go technology, the *${PROJECT_PATH_ROOT}/${PROJECT_GO}/.ci/setup_unix-any.sh* houses only the setup implementations for Go technology only.

All these CI recipes customization are done in their dedicated folders so the *AutomataCI/* main engine directory stays in-tact in case of future updates. In any cases, you should not be touching or editing any objects found in *AutomataCI/* directory at all. If you do, something is really wrong and you should reconsider or start asking questions.

9.1.2. CI Recipe File Naming Guideline

Generally speaking, the CI recipes are governed by their POSIX Shell and PowerShell dot import mechanism. Similarly to AutomataCI filesystem naming convention, a CI recipe shall be named following to the format tabulated in Table 8.

Table 8: A guideline for naming CI recipe

[_]{purpose}_{os}-{arch}.{extension}
LEGENDS: 1. **{ ... }** – compulsory field. 2. **[...]** – optional field.

Example:

1. **package_windows-any.ps1** – main Package CI job recipe for Windows OS environment.
2. **package_unix-any.sh** – main Package CI job recipe for UNIX OS environment.
3. **_package-msi_windows-any.ps1** – a broken-down package CI job recipe focusing only on MSI packaging for *package_windows-any.ps1* to import in.
4. **_package-msi_windows-any.sh** – a broken-down package CI job recipe focusing only on MSI packaging for *package_unix-any.sh* to import in.

Depending on the volume of a recipe, one can break it down to multiple files for maintainable sanity purposes. As shown in the example above, AutomataCI covers a large amount of packaging ecosystems so stuffing everything into a single large main recipe file is a guaranteed nightmare. Hence, the sensible strategy is to break it down (often has an underscore (_) prefix) like in (3) and (4) while the main recipe (1) and (2) is only responsible for importing the sub-recipes.

9.1.3. Job-Oriented Executions

Keep in mind that each CI job defines its own manner of executions and they're not the same with each other. The execution patterns are detailed in their respective sub-sections.

9.1.4. Concurrent/Parallel Implementations

While AutomataCI maintains the main CI job story-line, each underlying executions of the job can be executed concurrently or in parallel whenever possible. This speeds up the execution process and fully utilizes a given host machine's resources at a given time.

As of version 2.0.0, concurrent/parallel executions are already implemented in some CI jobs. They are detailed in their respective sub-sections.

9.2. ENV (Environment)

Environment (ENV) CI job operates by setting up the basic and required tools for executing all the CI jobs. These setup are usually common and made available across different OSes for a set of host machines.

9.2.1. Operating Mechanism

This CI Job executes in series. Parallel executions is unavailable.

9.2.2. No Customization

Due to simplicity sake, **this job does not offer any CI customization**. Should you need to setup more technologies, Setup CI Job recipe (detailed in later sub-section) would be the right place.

9.2.3. Sourcing Suppliers

Depending on OSes and the technology, AutomataCI engages:

1. **Homebrew** (https://docs.brew.sh/) for Linux & MacOS OSes; AND
2. **Chocolatey** (https://docs.chocolatey.org/en-us/default-chocolatey-install-reasoning) for Windows OS.

AutomataCI intentionally selects these 2 distribution suppliers mainly due to them not requiring root permission to mess with the host machine's internal system. Hence, this isolates the Host OS's internal system from a given CI deployment make the job a lot easier to work with.

9.2.4. Tools Procurement

Depending on CI job, these are the known tools ENV CI job will install into your host system.

1. **chocolatey** – Windows for Windows software procurement and chocolatey's nupkg testing.
2. **curl** – MacOS, Linux, & Windows for network interfacing.
3. **docker** – MacOS, Linux, & Windows for setting up Open Container Initiative's packager.
4. **dotnet** – MacOS (on request), Linux (on request), & Windows for accessing .NET libraries and executables.
5. **homebrew** – MacOS & Linux for software procurement.
6. **msitools** – MacOS & Linux for packaging Window's MSI format.
7. **osslsigncode** – MacOS & Linux for Windows executable notarization.
8. **reprepro** – MacOS & Linux for setting up debian package APT repository.
9. **signtool** – Windows OS for Windows executable notarization.
10. **wix4** – Windows OS for packaging Window's MSI format.

9.3. Setup

Setup CI job operates by setting up the project's technologies and virtual environment for the following operations and development. Unlike ENV CI job, the Setup CI Job focuses on establishing the actual workspace above the ENV CI job installation.

9.3.1. Operating Mechanism

This CI Job executes in series. Parallel executions are available but not recommended since setup sequences can break one another depending on OS and the type of software being installed (e.g. conflicting registry entries on Windows OS).

9.3.2. Customization

Setup CI job offers CI job recipe customization for both baseline (*$PROJECT_PATH_SOURCE*) and tech-specific (e.g. Go for *$PROJECT_GO*) sources. AutomataCI shall source their *.ci/setup_unix-any.sh* and *.ci/setup_windows-any.ps1* job recipes.

The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources ***common_unix-any.sh*** or ***common_windows-any.ps1*** (represented as "***common_{unix,windows}-any.{sh,ps1}***") automataCI execution.
4. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) tech-specific customized CI job recipe's ***$PROJECT_{TECH}/.ci/setup_unix-any.sh*** or ***$PROJECT_{TECH}/.ci/setup_windows-any.ps1*.**
5. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/setup_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/setup_windows-any.ps1*.**

9.3.3. Tools Procurement

Depending on CI job, these are the known tools ENV CI job will install into your host system.

1. **chocolatey** – Windows for Windows software procurement and chocolatey's nupkg testing.
2. **curl** – MacOS, Linux, & Windows for network interfacing.
3. **docker** – MacOS, Linux, & Windows for setting up Open Container Initiative's packager.
4. **dotnet** – MacOS (on request), Linux (on request), & Windows for accessing .NET libraries and executables.
5. **homebrew** – MacOS & Linux for software procurement.
6. **msitools** – MacOS & Linux for packaging Window's MSI format.
7. **osslsigncode** – MacOS & Linux for Windows executable notarization.
8. **reprepro** – MacOS & Linux for setting up debian package APT repository.
9. **signtool** – Windows OS for Windows executable notarization.
10. **wix4** – Windows OS for packaging Window's MSI format.

9.3.4. Default Tech-Specific Recipes

To ensure AutomataCI can provide warranted operating performances, AutomataCI provides its default tech-specific setup job recipes for reference or just a quick deployment. They are detailed in their sub-sections.

9.3.4.1. Angular (Javascript+Typescript) Framework

Should *$PROJECT_ANGULAR* is set to its source directory in *CONFIG.toml* (default is "*srcANGULAR/*"), AutomataCI shall setup 2 set of technologies for operating Angular framework (https://angular.io/):

1. NodeJS (https://nodejs.org/) JavaScript runtime engine; AND
2. Angular CLI (https://angular.io/cli).

Manually, the commands are the same as shown in Table 9.

Table 9: Angular Framework Manual Commands

```
# UNIX OS (Linux & MacOS)
$ brew install node
$ npm install -g @angular/cli

# WINDOWS OS
$ choco install node -y
$ npm install -g @angular/cli
```

Duly noted however, should AutomataCI detects any existing installation (supplied by host OS or already installed manually before execution), the Setup CI job shall automatically skip the duplicated installation.

The default Setup CI job shall setup the following toolkit:

1. *$PROJECT_PATH_ROOT/$PROJECT_PATH_TOOLS/node-engine* (runtime)

9.3.4.2. C Programming Language

Should *$PROJECT_C* is set to its source directory (default is "*srcC/*"), AutomataCI shall setup as many C compilers as its possibly can for catering smooth cross-compilations. Do note that setting up C Programming Language working environment can be very complicated depending on host machine OS, host machine CPU, the target device's OS, the target device's CPU architecture, and the target device's compilers' availability.

AutomataCI does offer embedded use cases (one of the edge case). It is the customer's duty to setup the compiler in the host machine. Known edge cases setup are documented in the following sub-sections.

9.3.4.2.1. From LINUX-AMD64 for AVR Microcontrollers

On *linux-amd64* host machine, the compilers for Microchip's AVR microcontrollers (ATMEGA series) can be supplied and installed via the Linux Host (Debian) OS native packaging system (e.g. APT) manually. The command are tabulated in Table 10.

Table 10: Manual commands for installing AVR C compilers for Linux Debian

```
$ su # enter root mode
$ apt install avr-libc avrdude binutils-avr gcc-avr srecord -y
$ apt install gdb-avr simulavr -y
```

9.3.4.2.2. From DARWIN-AMD64||ARM64 for AVR Microcontrollers

On *darwin-amd64* or *darwin-arm64* (Apple M-Series CPU) MacOS OSes, you may want to install external cross-compilers manually as shown in Table 11. See https://github.com/osx-cross/homebrew-avr#installing-homebrew-avr-formulae for more info.

Table 11: Manual commands for install AVR C Compilers for MacOS

```
$ brew tap osx-cross/avr
$ brew install avr-gcc
```

9.3.4.2.3. From Windows for Windows Microprocessor CPU Targets

On Windows host machine, when Chocolatey (https://community.chocolatey.org/) is used, AutomataCI can setup the packages shown in Table 12.

Table 12: Manual Commands for setting up Windows Microprocessor CPU target

```
$ choco install gcc-arm-embedded -y
$ choco install mingw -y
$ choco install emscripten -y # NOTE: there is a bug causing this to fail
```

Due to the extreme complexities operating C Programming Language on Windows OS, AutomataCI does not support C/C++ cross-compilation capabilities (as in FROM windows FOR elsewhere).

Also, AutomataCI does not support proprietary C/C++ development (e.g. C# and VS related products). They are likely comes with their own IDE system and AutomataCI encourages you to use them instead of AutomataCI.

9.3.4.2.4. From Linux-AMD64 for Known Microprocessor CPU Targets

AutomataCI supports all known cross-compilers made available via Linux Host's native software distributions. Alternatively, one can also supplement (although not recommended) missing compilers via the Homebrew ecosystem. For Debian OS, Table 13 tabulated all the manual commands for setting up the cross-compilers without altering the distribution architectures.

Table 13: Manual commands for installing C cross-compilers for Linux Debian

```
$ apt install build-essential crossbuild-essential-amd64 \
        crossbuild-essential-arm64 \
        crossbuild-essential-armel \
        crossbuild-essential-armhf \
        crossbuild-essential-i386 \
        crossbuild-essential-mips \
        crossbuild-essential-mips64 \
        crossbuild-essential-mips64el \
        crossbuild-essential-mips64r6 \
        crossbuild-essential-mips64r6el \
        crossbuild-essential-mipsel \
        crossbuild-essential-mipsr6 \
        crossbuild-essential-mipsr6el \
        crossbuild-essential-powerpc \
        crossbuild-essential-ppc64el \
        crossbuild-essential-s390x \
        -y

$ brew install gcc \
        aarch64-elf-gcc \
        arm-none-eabi-gcc \
        riscv64-elf-gcc \
        x86_64-elf-gcc \
        i686-elf-gcc \
        mingw-w64 \
        llvm \
        emscripten
```

9.3.4.3. Go Programming Language

Should *$PROJECT_GO* is set to its source directory in *CONFIG.toml* (default is "*srcGO/*"), AutomataCI shall download and setup Go compiler directly from its official website (https://go.dev/doc/install) and setup a virtual environment.

Duly noted however is that the "virtual environment" is an AutomataCI specific approach adopted from other programming languages practice for grouping all the packages used in the Project in 1 location for auditing and housekeeping purposes.

The default Setup CI job shall setup the following toolkit:

1. *$PROJECT_PATH_ROOT/$PROJECT_PATH_TOOLS/go-root* (compiler)
2. *$PROJECT_PATH_ROOT/$PROJECT_PATH_TOOLS/go-engine* (virtual environment)

9.3.4.4. Nim Programming Language

Should *$PROJECT_NIM* is set to its source directory in *CONFIG.toml* (default is "*srcNIM/*"), AutomataCI shall download and setup Nim compiler from Homebrew or Chocolatey and setup a virtual environment.

Duly noted however is that the "virtual environment" is an AutomataCI specific approach adopted from other programming languages practice for grouping all the packages used in the Project in 1 location for auditing and housekeeping purposes.

The default Setup CI job shall setup the following toolkit:

1. *$PROJECT_PATH_ROOT/$PROJECT_PATH_TOOLS/nim-engine* (virtual environment)

9.3.4.5. Python Programming Language

Should *$PROJECT_PYTHON* is set to its source directory in *CONFIG.toml* (default is "*srcPYTHON/*"), AutomataCI shall setup a Python virtual environment if Python3 is available and supplied by host machine.

Duly noted that AutomataCI only supports Python3 and above.

The default Setup CI job shall setup the following toolkit:

1. *$PROJECT_PATH_ROOT/$PROJECT_PATH_TOOLS/python-engine* (virtual environment)

9.3.4.6. Rust Programming Language

Should *$PROJECT_RUST* is set to its source directory in *CONFIG.toml* (default is "*srcRUST/*"), AutomataCI shall download and setup Rust compiler directly using a locally audited *rustup.sh* script from its official website (https://rustup.rs/) and setup a virtual environment.

Duly noted however is that the "virtual environment" is an AutomataCI specific approach adopted from other programming languages practice for grouping all the packages used in the Project in 1 location for auditing and housekeeping purposes.

The default Setup CI job shall setup the following toolkit:

1. *$PROJECT_PATH_ROOT/$PROJECT_PATH_TOOLS/rust-engine* (virtual environment)

9.4. Prepare

Prepare CI job operates by pulling all the technologies' dependencies into the project, allowing one to have the workspace ready for the following development or production build.

9.4.1. Operating Mechanism

This CI Job executes in series. Parallel executions are available but not recommended since setup sequences can break one another.

9.4.2. Customization

Setup CI job offers CI job recipe customization for both baseline (*$PROJECT_PATH_SOURCE*) and tech-specific (e.g. Go for *$PROJECT_GO*) sources. AutomataCI shall source their *.ci/prepare_unix-any.sh* and *.ci/prepare_windows-any.ps1* job recipes.

The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources ***common_unix-any.sh*** or ***common_windows-any.ps1*** (represented as "***common_{unix,windows}-any.{sh,ps1}***") automataCI execution.
4. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) tech-specific customized CI job recipe's ***$PROJECT_{TECH}/.ci/prepare_unix-any.sh*** or ***$PROJECT_{TECH}/.ci/prepare_windows-any.ps1*.**
5. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/prepare_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/prepare_windows-any.ps1*.**

9.4.3. Default Tech-Specific Recipes

To ensure AutomataCI can provide warranted operating performances, AutomataCI provides its default tech-specific prepare job recipes for reference or just a quick deployment. They are detailed in their sub-sections.

9.4.3.1. Angular (Javascript+Typescript) Framework

Should *$PROJECT_ANGULAR* is set to its source directory in *CONFIG.toml* (default is "*srcANGULAR/*"), AutomataCI shall pull the dependencies for the Project similar to executing the commands manually in Table 14.

Table 14: Angular Framework Dependencies Pull Commands

$ npm install

9.4.3.2. Go Programming Language

Should *$PROJECT_GO* is set to its source directory in *CONFIG.toml* (default is "*srcGO/*"), AutomataCI shall pull the dependencies for the Project similar to executing the commands manually in Table 15.

Table 15: Go Programming Language Dependencies Pull Commands

$ go get .

9.4.3.3. Nim Programming Language

Should *$PROJECT_NIM* is set to its source directory in *CONFIG.toml* (default is "*srcNIM/*"), AutomataCI shall pull the dependencies for the Project similar to executing the commands manually in Table 16.

Table 16: Nim Programming Language Dependencies Pull Commands

$ nimble refresh

9.4.3.4. Python Programming Language

Should *$PROJECT_PYTHON* is set to its source directory in *CONFIG.toml* (default is "*srcPYTHON/*"), AutomataCI shall pull the dependencies for the Project similar to executing the commands manually in Table 17.

Table 17: Python Programming Language Dependencies Pull Commands

$ pip install -r requirements.txt

9.4.3.5. Rust Programming Language

Should *$PROJECT_RUST* is set to its source directory in *CONFIG.toml* (default is "*srcRUST/*"), AutomataCI shall pull the dependencies for the Project similar to executing the commands manually in Table 18.

Table 18: Python Programming Language Dependencies Pull Commands

$ cargo fetch

9.5. Start

Start CI job operates by initializing the repository via AutomataCI for actual development. Depending on technologies, some requires and provides initialization instructions like Python with its *venv* sourcing commands.

9.5.1. Operating Mechanism

The default CI Job executes in series due to its lightweight role. Parallel executions are available.

9.5.2. Customization

Start CI job offers CI job recipe customization for both baseline (*$PROJECT_PATH_SOURCE*) and tech-specific (e.g. Go for *$PROJECT_GO*) sources. AutomataCI shall source their *.ci/start_unix-any.sh* and *.ci/start_windows-any.ps1* job recipes.

The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources ***common_unix-any.sh*** or ***common_windows-any.ps1*** (represented as "***common_{unix,windows}-any.{sh,ps1}***") automataCI execution.
4. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) tech-specific customized CI job recipe's ***$PROJECT_{TECH}/.ci/start_unix-any.sh*** or ***$PROJECT_{TECH}/.ci/start_windows-any.ps1*.**
5. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/start_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/start_windows-any.ps1*.**

9.5.3. Virtual Environment Initialization

AutomataCI usually by default setup all technologies' virtual environment which may contain one or more manual instructions. Hence, please read the on-screen instructions developed by the project maintainers & developers.

9.6. Test

Test CI job operates by initializing the project technologies' full test run designed by the project developers such as but not limited to unit tests and integrated tests.

9.6.1. Operating Mechanism

This CI Job executes in series. Parallel executions are available.

9.6.2. Customization

Setup CI job offers CI job recipe customization for both baseline (*$PROJECT_PATH_SOURCE*) and tech-specific (e.g. Go for *$PROJECT_GO*) sources. AutomataCI shall source their *.ci/test_unix-any.sh* and *.ci/test_windows-any.ps1* job recipes.

The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources ***common_unix-any.sh*** or ***common_windows-any.ps1*** (represented as "***common_{unix,windows}-any.{sh,ps1}***") automataCI execution.
4. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) tech-specific customized CI job recipe's ***$PROJECT_{TECH}/.ci/test_unix-any.sh*** or ***$PROJECT_{TECH}/.ci/test_windows-any.ps1*.**
5. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/test_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/test_windows-any.ps1*.**

9.6.3. Test Coverage Heatmap

AutomataCI supplied default test executions do facilitate test coverage heatmap files for pinpoint and effective testing whenever a technology has it. Should it do so, by default, the heatmap (usually in HTML format) file will be available at the following path:

```
$PROJECT_PATH_ROOT/$PROJECT_PATH_LOG/
```

9.6.4. Default Tech-Specific Recipes

To ensure AutomataCI can provide warranted operating performances, AutomataCI provides its default tech-specific test job recipes for reference or just a quick deployment. They are detailed in their sub-sections.

9.6.4.1. Angular (Javascript+Typescript) Framework

Should *$PROJECT_ANGULAR* is set to its source directory in *CONFIG.toml* (default is "*srcANGULAR/*"), AutomataCI shall first check for ***$PROJECT_SIMULATE_RELEASE_REPO*** environment variable set status before executing "*ng test*" command. This is mainly due to Karma test framework depends on a Blink-based browser (e.g. Chromium) to execute all its unit test suites.

Unlike guided by most Internet articles, AutomataCI discourages using PhantomJS to operate all the unit test codes mainly due to Angular's frontend art-directive nature where it depends on human artistic acceptances to realistically judge an art. Hence, AutomataCI shall simulate the test run instead.

AutomataCI uses the Angular's native Karma and Jasmine test framework to operate Angular test components. Hence, all existing knowledge and test suites can be directly reusable without additional learning. Learn more from: https://angular.io/guide/testing

9.6.4.2. C Programming Language

Should *$PROJECT_C* is set to its source directory (default is "*srcC/*"), AutomataCI shall use its internal infrastructure to facilitate unit testing for C Programming Language.

Unlike other modern technologies, C Programming Language does not have test infrastructure by default so AutomataCI has to facilitate one. AutomataCI challenges the fundamental of unit testing paradigm where each tech suite are statically compiled executable and execute separately by detecting "_test" filename suffix. Example, the for *greeter_any-any.c* library, its unit test codes can be *greeter_any-any_test.c*.

9.6.4.2.1. How It Works

Here's how AutomataCI works:

1. AutomataCI scans for all "***_test.c***" suffix source codes recursively across the project repository specifically *${PROJECT_PATH_ROOT}/${PROJECT_C}* directory.
2. Each of these test file are self-contained main file which then be compiled into its own executable based on the host machine.
3. AutomataCI then execute each of these executable.
    3.1. Should any of them fails, the test shall be concluded as failing entirely.

The build directory can also be manually verified and analyzed in the project workspace directory (*${PROJECT_PATH_ROOT}/${PROJECT_PATH_TEMP}/c-test_**).

IMPORTANT NOTE

Due to the way AutomataCI is designed, you should avoid testing cross-compilation build sequences in this Test CI job. Instead, develop the Build CI job properly and let Build CI reports those errors instead. The focus is on testing your C codes.

9.6.4.3. Go Programming Language

Should *$PROJECT_GO* is set to its source directory in *CONFIG.toml* (default is "*srcGO/*"), AutomataCI shall use "*go tool*" and "*go test*" features to mercilessly test all the applicable source codes specifically in *$PROJECT_PATH_ROOT/$PROJECT_GO/* directory.

Once completed, a test coverage heatmap generated by Go compiler is stored in the following directory:

```
${PROJECT_PATH_ROOT}/${PROJECT_PATH_LOG}/go-test-report/
```

9.6.4.4. Nim Programming Language

Should *$PROJECT_NIM* is set to its source directory in *CONFIG.toml* (default is "*srcNIM/*"), AutomataCI shall use its internal test infrastructure to facilitate unit testing for Nim Programming Language.

Although Nim provides *testament* package (https://nim-lang.org/docs/testament.html) for said task, AutomataCI keeps bumping into its errors and failures. AutomataCI challenges the fundamental of unit testing paradigm where each tech suite are statically compiled executable and execute separately by detecting "*_test*" filename suffix. Example, the for *greeter_any-any.nim* library, its unit test codes can be *greeter_any-any_test.nim*.

9.6.4.4.1. How It Works

Here's how AutomataCI works:

1. AutomataCI scans for all "***_test.nim***" suffix source codes recursively across the project repository specifically *${PROJECT_PATH_ROOT}/${PROJECT_NIM}* directory.
2. Each of these test file are self-contained main file which then be compiled into its own executable based on the host machine.
3. AutomataCI then execute each of these executable.
    3.1. Should any of them fails, the test shall be concluded as failing entirely.

The build directory can also be manually verified and analyzed in the project workspace directory (*${PROJECT_PATH_ROOT}/${PROJECT_PATH_TEMP}/nim-test_**).

IMPORTANT NOTE
Due to the way AutomataCI is designed, you should avoid testing cross-compilation build sequences in this Test CI job. Instead, develop the Build CI job properly and let Build CI reports those errors instead. The focus is on testing your Nim codes.

9.6.4.5. Python Programming Language

Should *$PROJECT_PYTHON* is set to its source directory in *CONFIG.toml* (default is "*srcPYTHON/*"), AutomataCI shall execute its test run using its *unittest* pip package alongside *coverage* pip package.

Once completed, a test coverage heatmap generated by *coverage* pip package is stored in the following directory:

```
${PROJECT_PATH_ROOT}/${PROJECT_PATH_LOG}/python-test-report/
```

9.6.4.5.1. Dependencies

As stated earlier, the AutomataCI supported default command depends on the following PyPi packages:

1. *coverage* – generate test coverage heatmap.

9.6.4.6. Rust Programming Language

Should *$PROJECT_RUST* is set to its source directory in *CONFIG.toml* (default is "*srcRUST/*"), AutomataCI shall execute test run using *grcov* and its *llvm-toolspreview* components, both installed via *cargo* and *rustup* respectively.

Although the facilities provided by Rust currently is quite messy, it is still capable of generating code coverage heatmap for developer to perform pinpoint accurate testing while executing the unit testing facilities.

AutomataCI executes this Test CI Job for Rust Programming Language based on the following documentations:

1. https://doc.rust-lang.org/rustc/instrument-coverage.html
2. https://github.com/mozilla/grcov
3. https://doc.rust-lang.org/rust-by-example/testing/unit_testing.html

Once completed, a test coverage heatmap generated by *gconv* and is stored in the following directory:

```
${PROJECT_PATH_ROOT}/${PROJECT_PATH_LOG}/rust-test-report/
```

9.6.4.6.1. Dependencies

As stated earlier, the AutomataCI supported default command depends on the following Cargo package and Rust toolkit:

1. *grcov* – generate test coverage heatmap.
2. *llvm-toolspreview* – internal test tools.

9.7. Materialize

Materialize CI job operates by building the repository's binary only for the host machine via AutomataCI. The effects and efforts is similar to Build CI Job (in next sub-section) except Build CI job builds for all targets using its cross-compilation capabilities. To avoid content duplication, please refer to Build CI job for specific build specifications.

The primary design for this CI Job is mainly to continue facilitate consistent build executions while being distributed alongside source codes packages (e.g. Git, Chocolatey, and Homebrew).

9.7.1. Operating Mechanism

The default CI Job executes in series due to its lightweight role. Parallel executions are available.

To avoid conflicting with Build CI Job recipe and also maintaining consistencies across UNIX OSes, Materialize CI Job shall output the built products into the following locations:

```
# Executables
${PROJECT_PATH_ROOT}/${PROJECT_PATH_BIN}/

# Libraries
${PROJECT_PATH_ROOT}/${PROJECT_PATH_LIB}/

# Documents
${PROJECT_PATH_ROOT}/${PROJECT_PATH_DOCS}/
```

9.7.2. Customization

Materialize CI job offers CI job recipe customization for both baseline (*$PROJECT_PATH_SOURCE*) and tech-specific (e.g. Go for *$PROJECT_GO*) sources. AutomataCI shall source their *.ci/materialize_unix-any.sh* and *.ci/materialize_windows-any.ps1* job recipes. The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources ***common_unix-any.sh*** or ***common_windows-any.ps1*** (represented as "***common_{unix,windows}-any.{sh,ps1}***") automataCI execution.
4. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) tech-specific customized CI job recipe's ***$PROJECT_{TECH}/.ci/materialize_unix-any.sh*** or ***$PROJECT_{TECH}/.ci/materialize_windows-any.ps1*.**
5. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/materialize_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/materialize_windows-any.ps1*.**

9.8. Build

Build CI job operates by building the project's products for all targets using its cross-compilation capabilities. The output is a production-ready version usually having debugging symbols stripped off and is highly optimized.

As the distribution ecosystem are moving towards server containerization and ease-of-use cases, **it's always advisable to produce a single binary executable and refrain from requesting your customers to sort out your product's dependencies**. It's your job, not theirs.

9.8.1. Operating Mechanism

This CI Job executes in parallel whenever available; in series as fallback.

The output directory for this CI Job is housed in:

```
# Executables & Libraries
${PROJECT_PATH_ROOT}/${PROJECT_PATH_BUILD}/

# Documents
${PROJECT_PATH_ROOT}/${PROJECT_PATH_DOCS}/
```

9.8.2. Customization

Setup CI job offers CI job recipe customization for both baseline (*$PROJECT_PATH_SOURCE*) and tech-specific (e.g. Go for *$PROJECT_GO*) sources. AutomataCI shall source their *.ci/build_unix-any.sh* and *.ci/build_windows-any.ps1* job recipes.

The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources ***common_unix-any.sh*** or ***common_windows-any.ps1*** (represented as "***common_{unix,windows}-any.{sh,ps1}***") automataCI execution.
4. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) tech-specific customized CI job recipe's ***$PROJECT_{TECH}/.ci/build_unix-any.sh*** or ***$PROJECT_{TECH}/.ci/build_windows-any.ps1*.**
5. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/build_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/build_windows-any.ps1*.**

9.8.3. Default Tech-Specific Recipes

To ensure AutomataCI can provide warranted operating performances, AutomataCI provides its default tech-specific setup job recipes for reference or just a quick deployment. They are detailed in their sub-sections.

9.8.3.1. Angular (Javascript+Typescript) Framework

Should *$PROJECT_ANGULAR* is set to its source directory in *CONFIG.toml* (default is "*srcANGULAR/*"), AutomataCI shall run "*ng build*" that produces the static website artifacts output files in the project directory.

Depending on the *angular.json* settings, specifically *architect.build.options.outputPath* field, referring to Table 19, the destination can be anywhere setup by the project (default is *$PROJECT_PATH_ROOT/public/* directory).

The output directory shall be *$PROJECT_PATH_DOCS* instead of *$PROJECT_PATH_BUILD* since it's a frontend static website builder.

Table 19: Angular Framework output path settings in *angular.json* config file

```
...
        "architect": {
                "build": {
                        "builder": "@angular-devkit/build-angular:browser",
                        "options": {
                                "outputPath": "../public",
                                "index": "src/index.html",
                                "main": "src/main.ts",
                                "polyfills": [
                                        "zone.js"
                                ],
...
```

9.8.3.2. C Programming Language

Should *$PROJECT_C* is set to its source directory (default is "*srcC/*"), AutomataCI shall use its operator functions to perform full-scale build without relying on another 3rd-party build tools like *Makefile*, *Cmake*, *make*, *Kconfig*, ..., etc. AutomataCI supports building the C executables and importable libraries with cross-platform capabilities by default.

These implementations are designed based on the following specifications:

1. https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
2. https://wiki.debian.org/Hardening
3. https://ntrs.nasa.gov/api/citations/19950022400/downloads/19950022400.pdf
4. https://www.kernel.org/doc/html/v4.10/process/coding-style.html
5. https://www.kernel.org/doc/html/v4.10/admin-guide/index.html
6. A book "Advanced Linux Programming" by Mark Mitchell, Jeffrey Oldham, Alex Samuel

9.8.3.2.1. Compile Functions

AutomataCI supplies *BUILD_Compile* (POSIX Shell) or *BUILD-Compile* (PowerShell) that offers all the required 3rd-party dependencies replacements. To build a C project involving multiple source codes, an example command is shown in Table 20.

Table 20: Build compile function offered by AutomataCI in POSIX Shell (top) and in PowerShell (bottom)

```
BUILD_Compile \
        “binary” \              # Param 1 – define the output type (‘binary’ or ‘library’)
        “linux” \               # Param 2 – define the target OS.
        “amd64” \               # Param 3 – define the target ARCH.
        “main-bin.txt” \        # Param 4 – define the relative path to the build list text file.
        “-Wall -Wextra …” \  # Param 5 – compiler arguments / settings.
        “gcc”                   # Param 6 – select compiler. Empty means automatic select
if [ $? -ne 0 -a $? -ne 10 ]; then
        # handle error
fi
```

```
$__process = BUILD-Compile `
        “binary”
        “windows”
        “amd64”
        “main-bin.txt”
        “-Wall -Wextra …”
        “cc”
if (($__process -ne 0) -and ($__process -ne 10)) {
        # handle error
}
```

This function shall return a code upon completion:

(a) **0** – successful run.

(b) **10** – not available. Recommended skipping.

(c) **Non-0** – error occurred.

To keep things maintainable since each compiler may or may not be compliant to C standards, the build variables can be used to facilitate a consistent and reusable settings across cross-compilers as shown in Table 21.

As shown in Table 21, by using an environment variables, one can also include additional libraries either statically or via linking.

Table 21: Tided up with build variables for cross compilations in POSIX Shell

```
...
SETTINGS_BIN="\
        -Wall \
        -Wextra \
        ...
        -Oz \
        -static \
        -L/path/to/your/custom/libs \
        -llibCustomA \
        -llibCustomB
"

BUILD_Compile \
        “binary” \
        “linux” \
        “amd64” \
        “main-bin.txt” \
        “$SETTINGS_BIN” \
        “gcc”
if [ $? -ne 0 -a $? -ne 10 ]; then
        # handle error
fi

...
```

9.8.3.2.2. Recommended Production Grade Compiler Configurations

The recommended production-grade configuration is as follows:

```
x86_64-linux-gnu-gcc -o {...}/lib/greeter_any-any.o -c {...}/libs/sample/greeter_any-any.c -Wall -Wextra - std=gnu99 -pedantic -Wstrict-prototypes -Wold-style-definition -Wundef -Wno-trigraphs -fno-strictaliasing -fno-common -fshort-wchar -fstack-protector-all -Werror-implicit-function-declaration -Wnoformat-security -pie -fPIE -Oz -static
```

9.8.3.2.3. File Structure and Organization

AutomataCI designed the C file structure organization in a way similar to other modern Programming languages where the header files (*.h*), external object files (*.o*), external library files (*.a* or *.dll*) for a single module is packed under a single directory. Each header files represents a package in the module and shall be made importable by the *main.c* file. Moreover, it's safer and easier to distribute a library or source codes by directory just in case.

Should any of the source file is a compiled object or a compiled library format, AutomataCI shall copy the compiled file over to the workspace during compilation phase for linking purposes.

Table 22 shows a minimal file structure and organization when using with AutomataCI.

Table 22: A minimal file structure and organization for AutomataCI

```
srcC/
├── automataCI.txt          # AutoamtaCI control config file.
├── libs
│   └── sample                      # a C package called 'sample'
│       ├── automataCI.txt
│       ├── entity_any-any.h
│       ├── greeter_any-any.c
│       ├── greeter_any-any.h
│       ├── greeter_any-any_test.c
│       └── location_any-any.h
└── main.c
```

9.8.3.2.4. AutomataCI Build Config File

Similar to *Kconfig* function adopted from Linux Kernel Project, AutomataCI relies on its build config file (replacing *Makefile* and *KConfig*) that lists all required files for the compilation and linking processes for your product. This config file is a text file where the filename can be anything (recommend: *automataCI.txt* to hint newcomer easily) as long as it is correctly being fed to the AutomataCI's compile function.

9.8.3.2.4.1. Config File Format

The text file's format is quite simple, as shown in Table 23.

Table 23: The format for the AutomataCI Build Config File

```
# DEPENDENCIES
any-any libs/sample/greeter_any-any.c

# CORE
any-any main.c
```

NOTE:

1. Each line (separated by newline (*n*)) represents statement that can be:
    1. A source file; OR
    2. A comment.
2. Anything beyond hash symbol (#) is a comment.
3. A source file has a relative pathing from the config file either with:
    1. *.c* extension – C source code requiring compilation to *.o* object file; OR
    2. *.o* extension – compiled object source file to be copied over.
4. A source file MUST leads with target platform (*{os}-{arch}*) configurations to let AutomataCI perform the required list of subjects based on the output target. To include in everything, use "*any-any*" as value.

9.8.3.2.5. AutomataCI Build Sequences

Since AutomataCI facilitates multiple build tools at the same time, it is vital to know its build sequences. At build, AutomataCI performs the following steps:

1. Setup *$PROJECT_PATH_ROOT/$PROJECT_PATH_TEMP/c-build** directory for workspace.
2. Parse the given AutomataCI build config file.
    2.1. Validate source file existence.
    2.2. Check source file's compatibility (*.c* or *.o* file extensions only).
    2.3. Register source file to:
        2.3.1. *$PROJECT_PATH_ROOT/$PROJECT_PATH_TEMP/c-build*/o-list.txt*
        2.3.2. *$PROJECT_PATH_ROOT/$PROJECT_PATH_TEMP/c-build*/sync.txt*
3. Build all object files in parallel using: *$PROJECT_PATH_ROOT/$PROJECT_PATH_TEMP/c-build*/sync.txt*
4. Link all object files based on *$PROJECT_PATH_ROOT/$PROJECT_PATH_TEMP/c-build*/o-list.txt* and export the final executable and linkable format (ELF) output file (*.elf* or *.exe*) or generate an ar type archived library file (*.a*).

Any custom post-processing execution (e.g. AVR hex dump from the ELF file) task shall be executed in the custom .CI script after the compile function.

9.8.3.2.6. Library Management

For those who are interested to build a dynamically linked executable against an internal library, it's highly recommended to:

1. build the library as an external product first; AND THEN
2. The main executable to import it in as a source file; AND THEN
3. The main executable is then built with the library file.

Otherwise, it is strongly advised to build the main executable from all available source codes and statically link them up together.

AutomataCI shall and always build the library file (.a) matching the AutomataCI build naming convention. It's the Packaging CI job duty to export into the library file to a compliant name like lib* prefix requirement in Linux OS system.

9.8.3.2.7. Cross-Compilations

While AutomataCI attempts to facilitate as much cross-compilation feature as possible in a simple approach for C programming language, there are limitations based on the availability in the host machine and the restrictions from the target system.

9.8.3.2.7.1. Linux (Debian-based)

Linux Debian cross compilation facilities enhanced by brew installer choice is among the best among all known development host system.

9.8.3.2.7.2. Darwin (MacOS)

Although the POSIX side of stuffs (e.g. libraries, ABI, etc) are similar, Apple's SDK and C libraries are notoriously complicated to be made available in the non-Apple operating system. The Clang compilation is supported but they are missing the Apple's SDK which then caused their cross-compilation to fail.

Moreover, the security notarization requirement further complicates the build process itself. Therefore, **like all Apple products, to build for Apple system, use a Mac OSX and its Xcode instead**.

Supports for Apple OSX to cross-compile for other OSes are limited compared to the Linux Debian counterparts (where it supports rare CPUs like mips and etc).

9.8.3.2.7.3. Windows

C in Windows only allows *MingW* compiler to compile for its own kind. There are no records of any successful cross-compilation to other OSes.

To avoid complications, AutomataCI *only* facilitates MingW compilation for Windows using MingW only.

9.8.3.3. Cargo Packager (Rust Programming Language)

Due to the possibility of Rust Programming Language may or may not be used in a project, AutomataCI relies on a placeholder signal file to package Rust's Cargo package.

Hence, in order to create one or more Cargo package, you need to generate an empty placeholder file as a build output for signaling Package CI Job (see later sub-section) to facilitate Cargo packaging.

The filename must consist of "**-cargo**" in the name section. The file is housed alongside other built outputs in $PROJECT_PATH_BUILD/ directory. Example:

$PROJECT_PATH_ROOT/$PROJECT_PATH_BUILD/myproduct**-cargo**_any-any

9.8.3.4. Chocolatey Packager

Due to the large probable combinations way of packaging a Chocolatey payload, AutomataCI employs a generic signal approach for developers to assemble the payloads accordingly.

Hence, in order to create one or more Chocolatey package, you need to generate an empty placeholder file as a build output for signaling Package CI Job (see later sub-section) to facilitate Chocolatey packaging.

The filename must consist of "**-chocolatey**" in the name section. The file is housed alongside other built outputs in $PROJECT_PATH_BUILD/ directory. Example:

$PROJECT_PATH_ROOT/$PROJECT_PATH_BUILD/myproduct**-chocolatey**_any-any

9.8.3.5. Go Programming Language

Should *$PROJECT_GO* is set to its source directory in *CONFIG.toml* (default is "*srcGO/*"), AutomataCI shall compile a full-static Go binary product by:

1. Disabling CGo (***CGO_ENABLED=0***); AND
2. Stripping all debugging symbols and trimpath; AND
3. PIE build mode (***-buildmode=pie***) enabled whenever available.

These output binary are fully static which can be operated even in an empty container (scratch).

However, this also comes at a cost: while AutomataCI loops through all the distribution list (*$ go tool dist list*), not all the target can be built without CGO. The list is available in the *build_[SYSTEM]* CI job script inside the *$PROJECT_GO* directory. This list shall be updated from time-to-time aligning to Go upstream updates.

9.8.3.6. Homebrew Packager

Due to the large probable combinations ways of packaging a Homebrew payload, AutomataCI employs a generic signal approach for developers to assemble the payloads accordingly.

Hence, in order to create one or more Homebrew package, you need to generate an empty placeholder file as a build output for signaling Package CI Job (see later sub-section) to facilitate Homebrew packaging.

The filename must consist of "**-homebrew**" in the name section. The file is housed alongside other built outputs in $PROJECT_PATH_BUILD/ directory. Example:

$PROJECT_PATH_ROOT/$PROJECT_PATH_BUILD/myproduct**-homebrew**_any-any

9.8.3.7. MSI Packager

Due to the large probable combinations ways of packaging a MSI payload, AutomataCI employs a generic signal approach for developers to assemble the payloads accordingly.

Hence, in order to create one or more MSI package, you need to generate an empty placeholder file as a build output for signaling Package CI Job (see later sub-section) to facilitate MSI packaging.

The filename must consist of "**-msi**" in the name section. The file is housed alongside other built outputs in $PROJECT_PATH_BUILD/ directory. Example:

$PROJECT_PATH_ROOT/$PROJECT_PATH_BUILD/myproduct**-msi**_any-any

9.8.3.8. Nim Programming Language

Should *$PROJECT_NIM* is set to its source directory in *CONFIG.toml* (default is "*srcNIM/*"), AutomataCI shall build Nim binary via its compiler utilizing C build compilers.

Unlike other languages, Nim is first compile into a targeted source code (in our case: C source code) internally and then uses the native compiler to compile into the executable or library (in our case: using C CI build job functions). Hence, there are multiple forms of output per builds:

1. The executable
2. The compiled source code (.c, .js, .objc, ...)

9.8.3.8.1.1. Default Nim Configurations

By default, AutomataCI builds with ORC garbage collection with extreme release optimization mode alongside C Programming Languages' build configurations.

Should you require customization, please refer: https://nim-lang.org/docs/nimc.html for detailed info.

9.8.3.8.1.2. WASM Compilation

At this point in time, WASM compilation cannot be confidently produce due to the absent of a proper Enscripten support (see: https://github.com/nim-lang/Nim/issues/8713). Until the support is ready, AutomataCI shall revisit once the issue is resolved.

9.8.3.8.1.3. AVR Compilation

At this point in time, AVR-GCC compilation failed in Nim while the source code and compilers are working in C Programming Language. It is suspected that the Nim standard libraries is at fault without factoring in Non-GC implementations and ecosystem (See: https://disconnected.systems/blog/nim-on-adruino/, https://github.com/zevv/nim-arduino, and https://github.com/dinau/nimOnAVR).

Hence, for the time being, AutomataCI shall not support AVR compilation by default.

9.8.3.8.1.4. Nim's Docgen Auto-Documentation

While Nim offers docgen capability to produce the package's documentations, at this point, it is not able to compose them properly without errors.

Hence, for the time being, AutomataCI shall not support native documentation compilation by default.

9.8.3.9. Python Programming Language

Should *$PROJECT_PYTHON* is set to its source directory in *CONFIG.toml* (default is "*srcPYTHON/*"), AutomataCI shall compile the Python Project using *pyinstaller* into a semi-static single file binary and *pdoc3* for generating documentations.

The build process is as follows:

1. Product created from *main.py*.
2. Compile into single semi-static binary using *pyinstaller* with --*onefile* argument.
3. Compile repository documentations using *pdoc3*.

9.8.3.9.1.1. Documentations

AutomataCI compiles the repository's documentations into a set of static website artifact that can be served with static sites service providers like Cloudflare Pages (https://pages.cloudflare.com/) or GitHub Pages (https://pages.github.com/).

9.8.3.9.1.2. Dependencies

In order for AutomataCI to function properly, the following *PyPi* packages shall be installed via the *requirements.txt*:

1. *pdoc3* – documentations
2. *pyinstaller* – for packaging Python files into single semi-linked binary.

9.8.3.10. Rust Programming Language

Should *$PROJECT_RUST* is set to its source directory in *CONFIG.toml* (default is "*srcRUST/*"), AutomataCI shall build the final executable based on the available optimizaton documented in the following sequences:

1. https://doc.rust-lang.org/cargo/reference/profiles.html
2. https://doc.rust-lang.org/nightly/rustc/platform-support.html

9.8.3.10.1.1. Default Configurations

As per AutomataCI mantra to produce single binary product for end-users, AutomataCI always prioritizes *musl* compilation for producing a static binary operating on its own. Unless otherwise required, customization can be done via the *build* or *materialize* CI job recipes.

9.8.3.10.1.2. Cross-Compilations

AutomataCI by default also supports cross-compilations based on the available compilers provided by the host system. The identified and supported cross-compiling targets are available in the following files:

```
automataCI/services/compilers/rust.{sh,ps1}
```

Specifically in these functions:

```
POSIX Shell:   RUST_Get_Build_Target "linux" "amd64"
PowerShell:    RUST-Get-Build-Target "windows" "amd64"
```

To procure a compiler, simply run the following command (can be added inside Setup CI job script) after setting Rust up entirely:

```
$ rustup target add x86_64-unknown-linux-musl
```

Due to the large amount of compiler choices in the catalog, AutomataCI only installs host's cross-compilers in the provided *srcRUST/* setup CI job directory.

9.9. Notarize

Notarize CI job operates by scanning through the *$PROJECT_PATH_ROOT/$PROJECT_PATH_BUILD/* directory for compatible executables and perform the required cryptographic signing and recognition process based on the targeted OS.

9.9.1. Operating Mechanism

This CI Job executes in series. Parallel executions are available but not recommended since some OS vendors has network throttling mechanism in-place.

9.9.2. Customization

Setup CI job offers CI job recipe customization for baseline (*$PROJECT_PATH_SOURCE*) source. AutomataCI shall source its *.ci/notarize_unix-any.sh* and *.ci/notarize_windows-any.ps1* job recipes.

The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources (and implicitly execute) ***notarize_unix-any.sh*** or ***notarize_windows-any.ps1*** (represented as "***notarize_{unix,windows}-any.{sh,ps1}***") AutomataCI execution.
4. ***notarize_{unix,windows}-any.{sh,ps1}*** sources baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/notarize_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/notarize_windows-any.ps1*** for notarize function.

9.9.3. Default Operating System Supports

To ensure AutomataCI can provide warranted operating performances, AutomataCI provides its default notarization job recipes for reference or just a quick deployment. They are detailed in their sub-sections.

9.9.3.1. Apple Ecosystems

AutomataCI only supports Apple ecosystems notarization only in Apple's development machine (e.g. Mac Mini, Macbook, etc) as always required by Apple. Cross-notarization is not possible at all thanks to Apple's vendor locked-in restrictions.

The documentations that were based on by AutomataCI are:

1. https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution/customizing_the_notarization_workflow
2. https://stackoverflow.com/questions/56890749/macos-notarize-in-script
3. https://github.com/thezik/notarizer/blob/main/notarizer

9.9.3.1.1. Supported Platforms

The current AutomataCI supported host platforms are shown in Table Text 24.

Text 24: Apple Ecosystem notarization support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	**N/A**	**N/A**	**N/A**
linux-amd64 (Red Hat - Fedora)	**N/A**	**N/A**	**N/A**
darwin-amd64 (Intel-based MacOS)	**Yes**	**No**	**N/A**
windows-amd64	**N/A**	**N/A**	**N/A**

NOTE:

1. "**N/A**" means "not applicable".

9.9.3.1.2. AutomataCI Apple Libraries

AutomataCI abstracted and provides some functions in the Apple Publisher libraries located in:

```
${LIBS_AUTOMATACI}/services/crypto/apple.sh
```

Known sign function is:

```
APPLE_Sign "./dest/directory/" "file.elf"
```

Where it uses *codesign*, *ditto*, and *xcrun* functions natively supplied by Apple MacOS.

Signing function availability can be checked via:

```
APPLE_Is_Available
if [ $? -ne 0 ]; then
        # not available. Handle error.
Fi
```

9.9.3.1.3. Required Secret Data

This function requires a number of secret environment variables supplied either by setting it directly or via *SECRETS.toml* file. They are:

1. **APPLE_DEVELOPER_ID** – The developer ID used for notarization process.
2. **APPLE_KEYCHAIN_PROFILE** – The keychain access used for accessing notarization credentials.

9.9.3.2. Microsoft Ecosystems

AutomataCI supports Microsoft ecosystems notarization using various tools. When operating on Windows OS, AutomataCI uses its native *signtool.exe* program for signing; when operating on UNIX OS however, AutomataCI uses *osslsigncode* 3rd-party program to perform the signing.

The documentations that were based on by AutomataCI are:

1. https://learn.microsoft.com/en-us/dotnet/framework/tools/signtool-exe
2. https://github.com/mtrojnar/osslsigncode

9.9.3.2.1. Supported Platforms

The current AutomataCI supported host platforms are shown in Table Text 25.

Text 25: Apple Ecosystem notarization support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	**Yes**	**No**	**N/A**
linux-amd64 (Red Hat - Fedora)	**Yes**	**No**	**N/A**
darwin-amd64 (Intel-based MacOS)	**Yes**	**No**	**N/A**
windows-amd64	**Yes**	**No**	**N/A**

NOTE:

1. "**N/A**" means "not applicable".

9.9.3.2.2. AutomataCI Microsoft Libraries

AutomataCI abstracted and provides some functions in the Microsoft Publisher libraries located in:

```
${LIBS_AUTOMATACI}/services/crypto/microsoft.sh
${LIBS_AUTOMATACI}/services/crypto/microsoft.ps1
```

Known sign function is:

MICROSOFT_Sign “./dest/directory/” “file.elf” “Entity Name” “Entity Website”

Where it uses *osslsigncode* or *signtool* program.

Signing function availability can be checked via:

```
MICROSOFT_Is_Available
if [ $? -ne 0 ]; then
        # not available. Handle error.
Fi
```

9.9.3.2.3. Required Secret Data

This function requires a number of secret environment variables supplied either by setting it directly or via *SECRETS.toml* file. They are:

1. **MICROSOFT_CERT** – the full file-path to the cert file. On UNIX system, the cert format can be normal cert, SPC, or PCKS12 contained. On Windows however, only PCKS12 is accepted.
2. **MICROSOFT_CERT_HASH** – the required checksum algorithm specification. Can be: ‘SHA256', 'MD5', 'SHA1', 'SHA2', 'SHA384', and 'SHA512'. When in doubt, use SHA256.
3. **MICROSOFT_CERT_TYPE** – the required cert type specification only in UNIX system. Values can be: ‘CERT’, ‘SPC’, and ‘PCKS12’.
4. **MICROSOFT_CERT_TIMESTAMP** – the required Timestamp source usually an URL provided by a certificate authority.
5. **MICROSOFT_KEYFILE** – optional file used only in UNIX system when ‘CERT’ or ‘SPC’ cert types are used. It holds the private key of the certificate.
6. **MICROSOFT_CERT_PASSWORD** – The required password to decrypt the cert file.w

9.10. Package

Package CI job operates by scanning through the *$PROJECT_PATH_ROOT/$PROJECT_PATH_BUILD/* directory for compatible subject and package them into the industrially known distribution channels such as but not limited to Windows Store, Debian APT ecosystem, Red Hat's DEF ecosystem, Red Hat's Flatpak ecosystem, Apple's Brew ecosystem, CI friendly *.tar.xz* or *.zip* archives ecosystem, etc with security protocols and verifiable integrity.

9.10.1. Operating Mechanism

This CI Job executes in parallel by default and in series as fallback or incompatible processes.

Unlike other CI jobs, **Package CI job works differently where the customization are supplying the content assembling function instead of overriding an execution**. In the event where a tech-specific CI job recipe provided a same content assembling function of its predecessor, it shall be overridden with the latest version being used.

Hence, in any case, ***to avoid overlapping implementations and chaotic duplication, it's highly recommended to keep all the package CI job algorithms inside the baseline job recipe only***.

Although tech-specific CI job recipes are made available, they are meant for one to merge back via overrides into the baseline job recipes. Only the owner of the project knows what technologies to be used and AutomataCI shall not dictate but to facilitate all known and possible use cases.

9.10.2. Cryptography Signing

It's duly noted that some ecosystems require cryptography notorization such as but not limited to GPG signing for .deb and .rpm package types. *If there are such a need, it is always advisable to assemble all the built binary files in the right location and package it locally rather than relying on 3rd-party CI service provider*.

9.10.3. Limited Customization

Package CI job offers limited CI job recipe customization for both baseline (*$PROJECT_PATH_SOURCE*) and tech-specific (e.g. Go for *$PROJECT_GO*) sources. AutomataCI shall source their *.ci/package_unix-any.sh* and *.ci/package_windows-any.ps1* job recipes.

The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources ***package_unix-any.sh*** or ***package_windows-any.ps1*** (represented as "***package_{unix,windows}-any.{sh,ps1}***") automataCI execution.
4. ***package_{unix,windows}-any.{sh,ps1}*** sources common content assembling function from baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/package_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/package_windows-any.ps1*.**
5. ***package_{unix,windows}-any.{sh,ps1}*** sources tech-specific content assembling function from customized CI job recipe's ***$PROJECT_{TECH}/.ci/package_unix-any.sh*** or ***$PROJECT_{TECH}/.ci/package_windows-any.ps1*.**
6. ***package_{unix,windows}-any.{sh,ps1}*** schedules both parallel and serial executions for all probable combinations of packages.
7. ***package_{unix,windows}-any.{sh,ps1}*** prepares all workspace (actual, output, and log).
8. ***package_{unix,windows}-any.{sh,ps1}*** performs run for all parallel executions.
9. ***package_{unix,windows}-any.{sh,ps1}*** performs run for all serial executions.

9.10.4. Default Operating System Supports

To ensure AutomataCI can provide warranted operating performances, AutomataCI provides its default job recipes for reference or just a quick deployment. They are detailed in their sub-sections.

9.10.4.1. Archives Packages (*.tar.gz || .zip*)

AutomataCI supports primitive package archiving using *.tar.xz* or *.zip* archivers depending on the target OS (Windows is the only one using *.zip* archive format). By default, AutomataCI deploys the maximum compression performance (e.g. level 9 for XZ compressor) to ensure the output can be stored in long-term storage elsewhere and be energy efficient during transit.

This type of package is heavily used by continuous integration ecosystem and acting as a fallback.

Starting from version 1.7.0, AutomataCI package Archive type in parallel execution.

9.10.4.1.1. Supported Platforms

The current AutomataCI supported host platforms are shown in Table 26.

Table 26: Archive packaging support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	**Yes**	**Yes**	**Yes**
linux-amd64 (Red Hat - Fedora)	**Yes**	**No**	**Yes**
darwin-amd64 (Intel-based MacOS)	**Yes**	**Yes**	**Yes**
windows-amd64	**Yes**	**Yes**	**Yes**

9.10.4.1.2. Content Assembling Function

The content assembling function is shown in Table 27.

Table 27: the content assembly function symbol for archives packager

```
# POSIX Shell
PACKAGE_Assemble_ARCHIVE_Content() {
        _target="$1"
        _directory="$2"
        _target_name="$3"
        _target_os="$4"
        _target_arch="$5"

        ...
}
```

```
# PowerShell
PACKAGE-Assemble-ARCHIVE-Content {
        param(
                [string]$_target,
                [string]$_directory,
                [string]$_target_name,
                [string]$_target_os,
                [string]$_target_arch
        )

        ...
}
```

The objectives are:

1. To assemble the contents (files & directories) for archive packager to *.tar.gz* or *.zip* it.
    1. Please consider the user experience when your customer opens up or unpack the package.

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.10.4.1.3. Required Files

This packager does not impose any required files so long the directory is not empty.

9.10.4.1.4. Testing Package

To test an archive package, simply try to unpack it like an end-user. Example, for POSIX Shell:

```
$ tar -xvf “[NAME].tar.gz” .
$ unzip “[NAME].zip” -d .
```

9.10.4.2. Cargo Packages (*Rust Programming Language*)

AutomataCI supports Cargo packages for Rust Programming Language native ecosystem when a signaling placeholder file with "**-cargo**" keyword in its name (e.g. "myproduct**-cargo**_any-any") is detected. The supporting documentations AutomataCI based on are as follows:

1. https://doc.rust-lang.org/cargo/reference/registries.html
2. https://doc.rust-lang.org/cargo/reference/manifest.html
3. https://doc.rust-lang.org/cargo/reference/publishing.html
4. https://doc.rust-lang.org/cargo/guide/cargo-toml-vs-cargo-lock.html

Starting from version 2.0.0, AutomataCI package Cargo type in parallel execution.

9.10.4.2.1. Supported Platforms

The current AutomataCI supported host platforms are shown in Table 28.

Table 28: Cargo packaging support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	**Yes**	**Yes**	**Yes**
linux-amd64 (Red Hat - Fedora)	**Yes**	**No**	**Yes**
darwin-amd64 (Intel-based MacOS)	**Yes**	**Yes**	**Yes**
windows-amd64	**Yes**	**Yes**	**Yes**

9.10.4.2.2. Content Assembling Function

The content assembling function is shown in Table 29.

Table 29: the content assembly function symbol for Cargo packager

```
# POSIX Shell
PACKAGE_Assemble_CARGO_Content() {
        _target="$1"
        _directory="$2"
        _target_name="$3"
        _target_os="$4"
        _target_arch="$5"

        ...
}
```

```
# PowerShell
PACKAGE-Assemble-CARGO-Content {
        param(
                [string]$_target,
                [string]$_directory,
                [string]$_target_name,
                [string]$_target_os,
                [string]$_target_arch
        )

        ...
}
```

The objectives are:

1. to assemble a buildable source codes consumable by Cargo and its build tools; AND
2. to script the *${_directory}/Cargo.toml* crate file (specifically the *[package]* fields) based on the AutomataCI configurations used by Cargo externally at the end user side.

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.10.4.2.3. Required Files

This packager has the following required files:

1. *Cargo.toml* – template for generating the final *Cargo.toml* file.

IMPORTANT NOTE

The original *Cargo.toml* file (residing in *$PROJECT_RUST* directory) is used for operating Rust crate as usual *with the [package] fields ignored*. **A special commented flag (# [AUTOMATACI BEGIN]) MUST be specified to tell AutomataCI to append anything lines after it during the *Cargo.toml* file generation**.

The sole reason for regenerating *Cargo.toml* file during packaging is to make sure the *[package]* fields contain the **CONFIG.toml** project metadata values consistently across all other technological implementations.

AutomataCI shall run Cargo to validate the assembled package before staging it for Release CI job. Should there be any errors reported by Cargo, it should be resolved using your Rust programming language domain knowledge.

9.10.4.2.4. Testing Package

At the moment, Rust does not provide a testable way to test the Cargo package. The current practice is to push it against a test package in the registry and manually test it.

9.10.4.3. Chocolatey Packages

AutomataCI supports Chocolatey packages with "**-chocolatey**" keyword in its name (e.g. "myproduct**-chocolatey**_any-any") is detected. The supporting documentations AutomataCI based on are as follows:

1. https://docs.chocolatey.org/en-us/create/create-packages
2. https://learn.microsoft.com/en-us/nuget/reference/nuspec
3. https://blog.chocolatey.org/2016/01/create-chocolatey-packages/
4. https://learn.microsoft.com/en-us/nuget/
5. https://learn.microsoft.com/en-us/nuget/quickstart/create-and-publish-a-package-using-the-dotnet-cli
6. https://learn.microsoft.com/en-us/nuget/create-packages/creating-a-package
7. https://community.chocolatey.org/courses/creating-chocolatey-packages/summary-rules-and-guidlines
8. https://stackoverflow.com/questions/14329243/examine-contents-of-a-nuget-package
9. https://blog.chocolatey.org/2016/01/host-your-own-server/
10. https://docs.chocolatey.org/en-us/features/host-packages#local-folder-permissions
11. https://docs.chocolatey.org/en-us/features/host-packages#local-folder-or-share-structure

Starting from version 2.0.0, AutomataCI package Chocolatey type in parallel execution.

9.10.4.3.1. How it Works

AutomataCI counts on the fact that Chocolatey relies heavily on .Net framework's nupkg specification and some heavy batteries just to create a

> "*... a fancy version of a zip file that knows about metadata, versioning and dependencies related to underlying software, plus optional automation scripts that are run during installation, upgrade and uninstallation of the package ...*" — by Chocolatey.

Hence, AutomataCI itself uses the understanding of the Chocolatey & NuGet package algorithms and developed its own compiler for cross-packaging sake. AutomataCI scripts all the required files and zip it accordingly.

By default, AutomataCI recommends only package a compilable source codes alongside AutomataCI tool for end-user's compilation. This is mainly to handle 2 problems:

1. Microsoft Signing Security Requirement (https://learn.microsoft.com/en-us/windows/win32/seccrypto/signtool) — ensures the built binary is code-signed locally at the end user side of things; AND
2. Fulfilling Chocolatey's basic requirement.

Should the package delivers pre-compiled binaries, it's entirely your responsibility to get the binary both code-signed and notarized in your build system. Otherwise, should the product is accepted by the Chocolatey Core team, the package is published. For GUI and etc, AutomataCI highly recommends to split it into a separate package since not everyone is using GUI interface.

9.10.4.3.2. Supported Platforms

The current AutomataCI supported host platforms are shown in Table 30.

Table 30: Chocolatey packaging support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	**Yes**	**Yes**	**Yes**
linux-amd64 (Red Hat - Fedora)	**Yes**	**No**	**Yes**
darwin-amd64 (Intel-based MacOS)	**Yes**	**Yes**	**Yes**
windows-amd64	**Yes**	**Yes**	**Yes**

9.10.4.3.3. Content Assembling Function

The content assembling function is shown in Table 31.

Table 31: the content assembly function symbol for Chocolatey packager

```
# POSIX Shell
PACKAGE_Assemble_CHOCOLATEY_Content() {
        _target="$1"
        _directory="$2"
        _target_name="$3"
        _target_os="$4"
        _target_arch="$5"

        ...
}
```

```
# PowerShell
PACKAGE-Assemble-CHOCOLATEY-Content {
        param(
                [string]$_target,
                [string]$_directory,
                [string]$_target_name,
                [string]$_target_os,
                [string]$_target_arch
        )

        ...
}
```

The objectives are:

1. to assemble a build-able source codes consumable by Chocolatey and its build tools inside "*${_directory}/Data/*" sub-directory; AND
2. to script the required PowerShell scripts:
    1. *${_directory}/tools/chocolateyinstall.ps1*; AND
    2. *${_directory}/tools/chocolateyinstall.ps1*; AND
    3. *${_directory}/tools/chocolateyBeforeModify.ps1*

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.10.4.3.4. Required Files

This packager has the following required files:

1. **${_directory}/tools/chocolateyinstall.ps1** – the install execution instructions in PowerShell format.
2. **${_directory}/tools/chocolateyinstall.ps1** – the uninstall execution instructions in PowerShell format.
3. **${_directory}/tools/chocolateyBeforeModify.ps1** – the pre-configurations (e.g. disabling services and etc) execution instructions before executing any install/uninstall PowerShell scripts above.

Should the required files are missing, AutomataCI shall fail the entire Package CI Job run.

9.10.4.3.5. Testing Package

On Windows host machine, when *choco* is available, AutomataCI does execute a localized install and uninstall test run to ensure its built package is working properly as instructed by its official documentation (https://docs.chocolatey.org/en-us/create/create-packages#testing-your-package).

On UNIX OS however, there is no way to test it aside manual review since Chocolatey is a Windows-only ecosystem.

9.10.4.4. DEB Packages (*Debian OS*)

AutomataCI supports DEB packages by default using its own packager based on Debian specification. This approach was required mainly because the toolkit provided by Debian OS is too massive and some are deviated from the specifications (e.g. excessive checking).

Although AutomataCI uses its own compiler, the output shall always be compliant with upstream. you're still required to learn through the specifications (at least binary package) shown above before proceeding to construct the job recipe. The supporting documentations AutomataCI based on are as follows:

1. https://www.debian.org/doc/debian-policy/index.html
2. https://manpages.debian.org/unstable/dpkg-dev/deb.5.en.html
3. https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html

AutomataCI prioritizes binary package build since the source codes are usually distributed using git control ecosystem. This removes any duplication related to the project and focus on customer delivery instead.

Windows OS is not supported mainly due to the lack of UNIX permission manipulation (e.g. *chmod* or *chown* commands).

Starting from version 2.0.0, AutomataCI package DEB type in parallel execution.

9.10.4.4.1. Supported Platforms

The current AutomataCI supported host platforms are shown in Table 32.

Table 32: DEB packaging support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	**Yes**	**Yes**	**Yes**
linux-amd64 (Red Hat - Fedora)	**Yes**	**No**	**Yes**
darwin-amd64 (Intel-based MacOS)	**Yes**	**Yes**	**Yes**
windows-amd64	**No**	**No**	**N/A**

NOTE:

1. "**N/A**" – means "Not Available".

9.10.4.4.2. Content Assembling Function

The content assembling function is shown in Table 33.

Table 33: the content assembly function symbol for DEB packager

```
# POSIX Shell
PACKAGE_Assemble_DEB_Content() {
        _target="$1"
        _directory="$2"
        _target_name="$3"
        _target_os="$4"
        _target_arch="$5"

        ...
}
```

```
# PowerShell
PACKAGE-Assemble-DEB-Content {
        param(
                [string]$_target,
                [string]$_directory,
                [string]$_target_name,
                [string]$_target_os,
                [string]$_target_arch
        )

        ...
}
```

The *$_directory* variable should point to the workspace directory containing 2 important directories:

1. **control/** – housing the control components of the *.deb* package; AND
2. **data/** – housing the data components of the *.deb* package.

The objectives are:

1. to assemble the "to be installed" file structure in the **data/** directory complying to Filesystem Hierarchy Standard; AND
2. to assemble any maintainer scripts (if needed) in the **control/** directory.

For example:

1. the provided *$_target* variable that is pointing to the currently detected binary executable. It is usually being copied to **${directory}/data/usr/local/bin/** directory.
    1. This means that the package shall the target program into /usr/local/bin/ when the package is installed by the customer.

AutomataCI provides sufficient utilities to create all the required files. It's entirely your duty to assemble all the files in their respective location and only create the **control/control** file as the last step due to its requirement of calculating **data/** directory disk space consumption.

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.10.4.4.3. Required Files

This packager has the following required files:

1. data/usr/share/docs/${PROJECT_SKU}/changelog.gz
   OR
   data/usr/local/share/docs/${PROJECT_SKU}/changelog.gz
2. data/usr/share/docs/${PROJECT_SKU}/copyright.gz
   OR
   data/usr/local/share/docs/${PROJECT_SKU}/copyright.gz
3. data/usr/share/man/man1/${PROJECT_SKU}.1.gz
   OR
   data/usr/local/share/man/man1/${PROJECT_SKU}.1.gz
4. control/md5sum
5. control/control

These files follow strict format and content as specified in the Debian manual (especially *control/control* file).

9.10.4.4.4. Maintainers' Scripts

As frown by the Debian specification when used for installing files and directories, maintainers' scripts are only used when required. Manual efforts can cause unwanted and unclean executions at the end-user sides.

These optional scripts are usually used for emergency and adaptive patching, services (e.g. systemd, cron, nginx, etc) signaling and control before or after an install or uninstall process.

Supported Maintainers' scripts are:

1. **control/preinst** – commands before install
2. **control/postinst** – commands after uninstall
3. **control/prerm** – commands before uninstall
4. **control/postrm** – commands after uninstall

When in doubt, use *post[ACTION]* instead.

9.10.4.4.5. AutomataCI's Copyright.gz Generative Function

AutomataCI constructs the copyright.gz file by generating the license stanza and then appending the copyright text file located here:

```
${PROJECT_PATH_SOURCE}/licenses/deb-copyright
```

You should construct the license file as it is. Please keep in mind that this file is heavily governed by Debian Policy Manual and you should at least go through the specification for binary package described here: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/.

GENTLE REMINDER

Only need to generate the body of the file as the automation will generate the license stanza automatically. If there is a need to modify the process, consider overriding the output manually.

9.10.4.4.6. AutomataCI's Long Description Generative Function

Due to the long description length ambiguity, AutomataCI relies on external data file for generating the DEB's control file's *Description:* long data field. The file is located at:

```
${PROJECT_PATH_RESOURCES}/docs/ABSTRACTS.txt
```

Please keep in mind that this data file is also shared by other packagers or release processes for maintaining consistencies.

The file **MUST** comply to:

1. Max column length of 69 characters per line; AND
2. Strictly text-only; AND
3. UTF-8 but do only use ASCII characters for maximum backward compatibility (means English only).

9.10.4.4.7. Testing Package

To test a built *.deb* package, simply use the following commands (when available):

```
$ dpkg-deb --contents "[NAME].deb"
$ dpkg-deb --info "[NAME].deb"
```

9.10.4.5. Flatpak Packages (*Red Hat*)

AutomataCI supports Red Hat's Flatpak (also known as "Flatpak") packages by default using Flatpak provided builder program. Flatpak is a cross-Linux platform equipped with sandbox capabilities for securely and peacefully distributing applications exclusively across the Linux OSes; both Red Hat based and Debian based alike.

The supporting documentations AutomataCI based on are as follows:

1. https://docs.flatpak.org/en/latest/introduction.html
2. https://www.freedesktop.org/software/appstream/docs/chap-Metadata.html
3. https://specifications.freedesktop.org/menu-spec/latest/apa.html

Due to the reliance of Linux kernel by the Flatpak builder, both Windows OS and Mac OS are not supported.

Starting from version 2.0.0, AutomataCI package Flatpak type in parallel execution.

9.10.4.5.1. Supported Platforms

The current AutomataCI supported host platforms are shown in Table 34.

Table 34: Flatpak packaging support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	**Yes**	**Yes**	**Yes**
linux-amd64 (Red Hat - Fedora)	**Yes**	**No**	**Yes**
darwin-amd64 (Intel-based MacOS)	**N/A**	**N/A**	**N/A**
windows-amd64	**N/A**	**N/A**	**N/A**

NOTE:

1. "**N/A**" – means "Not Available".

9.10.4.5.2. Content Assembling Function

The content assembling function is shown in Table 35.

Table 35: the content assembly function symbol for Flatpak packager

```
# POSIX Shell
PACKAGE_Assemble_FLATPAK_Content() {
        _target="$1"
        _directory="$2"
        _target_name="$3"
        _target_os="$4"
        _target_arch="$5"

        ...
}
```

```
# PowerShell
PACKAGE-Assemble-FLATPAK-Content {
        param(
                [string]$_target,
                [string]$_directory,
                [string]$_target_name,
                [string]$_target_os,
                [string]$_target_arch
        )

        return 10 # not available in Windows OS
}
```

The objectives are:

1. to assemble the “to be installed” file structure into this directory; AND
2. to script the *${_directory}/manifest.yml* file; AND
3. to script the *${_directory}/appdata.xml* file.

For example:

1. the provided *$_target* variable that is pointing to the currently detected binary executable. It is usually being copied to ***${directory}/bin/*** directory.

Hence, it is recommended to just focus on constructing the package’s data path and leave the rest of the required files to the AutomataCI generative function.

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.10.4.5.3. Required Files

This packager has the following required files:

1. **manifest.yml** – instructions file for *flatpak-builder* to package.
2. **appdata.xml** – metadata for promoting your package visually in marketplace.
3. **icon.svg** – trademark for promoting your package visually in the market stated in *appdata.xml*.
4. **icon-48x48.png** – trademark for promoting your package visually in the market stated in *appdata.xml*.
5. **icon-128x128.png** – trademark for promoting your package visually in the market stated in *appdata.xml*.

These files follow strict format and content as specified in the Flatpak specification. Refer:

1. https://docs.flatpak.org/en/latest/freedesktop-quick-reference.html
2. https://specifications.freedesktop.org/icon-theme-spec/icon-theme-spec-latest.html

If AutomataCI fails to detect all the require files, it shall fail the entire Package CI Job.

9.10.4.5.4. Marketing Screenshots Artifacts

The screenshots for the *appdata.xml* file in accordance to the specification here (refer: https://www.freedesktop.org/software/appstream/docs/chap-Metadata.html#tag-screenshots) are best hosted elsewhere and back-linked into the XML data file. There are no signs of the screenshots can be loaded from the package internally.

9.10.4.5.5. Repository Branch Management

As per documentation (see: https://docs.flatpak.org/en/latest/using-flatpak.html#identifier-triples), AutomataCI treats each of the Flatpak branches as the supporting architecture. Hence, should your project supports multiple architectures by default, your Flatpak repository should checkout the branches in accordance to the CPU type.

The default-branch is set to "**any**".

9.10.4.5.6. Sandbox Permission

AutomataCI relies on an external text file to populate the package's sandbox permission. The file is located at:

```
${PROJECT_PATH_SOURCE}/packages/flatpak.perm
```

The values are exactly from the documentation. Please refer: https://docs.flatpak.org/en/latest/manifests.html#finishing

9.10.4.5.7. Simultaneous Repository Release

Due to *flatpak-builder* design, in order to package Flatpak type, it must be released to a file-based repository at the same time. Hence, its Release CI job is simultaneously executed and the private repository is saved at:

```
${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}/flatpak
```

where:

1. *${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}* is actually a local version of *${PROJECT_STATIC_REPO}* URL.

9.10.4.5.8. Directory-Based Output

Due to *flatpak-builder* design, all successful Flatpak packages are directory-based housing the required files for *flatpak-builder* to operate.

You're free to inspect the output directories but leave them as it is to avoid breaking later CI job.

9.10.4.5.9. Single Bundle

To ensure a fail-safe is available, AutomataCI automatically exports the single bundle format (refer: https://docs.flatpak.org/en/latest/single-file-bundles.html) that allows user to manually import the software without needing to track an upstream repository. This bundle file is included in the *$PROJECT_PATH_ROOT/$PROJECT_PATH_PKG/* directory ending with *.flatpak* file extension as required.

9.10.4.5.10. Testing Package

One can use *flatpak* to install the package locally. The command is as follows:

```
$ flatpak install --user [NAME].flatpak
```

AutomataCI can react based on the direct feedback from *flatpak-builder* in its building operation so manual testing is not required.

9.10.4.6. Homebrew Packages

AutomataCI supports Homebrew packages with "**-homebrew**" keyword in its name (e.g. "myproduct**-homebrew**_any-any") is detected. The supporting documentations AutomataCI based on are as follows:

1. https://docs.brew.sh/
2. https://docs.brew.sh/Taps
3. https://docs.brew.sh/Formula-Cookbook
4. https://github.com/Homebrew/homebrew-core/tree/master
5. https://brew.sh/2020/11/18/homebrew-tap-with-bottles-uploaded-to-github-releases/

Starting from version 2.0.0, AutomataCI package Homebrew type in parallel execution.

9.10.4.6.1. How it Works

AutomataCI counts on generating the Homebrew usable YAML manifest easily with some placeholder fields to mitigate Homebrew's "chicken and egg" problems.

Homebrew itself is somehow complicated when it comes to being automated by any CI pipelines mainly due to Apple's security requirements and its rather poor management over handling its vast coverage of development permutation possibilities. Moreover, the introduction of its own brewing terminologies makes the technical comprehension notoriously confusing albeit being flamboyant and fancy. After careful analysis over Homebrew's ecosystem, it's safer to let Homebrew to have its own dedicated package archive for distributions.

By default, AutomataCI recommends only package a compatible source codes alongside AutomataCI tool for end-user's compilation. This is mainly to handle 2 problems:

1. Microsoft Signing Security Requirement (https://learn.microsoft.com/en-us/windows/win32/seccrypto/signtool) — ensures the built binary is code-signed locally at the end user side of things; AND
2. Fulfilling Homebrew's basic requirement.

Should the package delivers pre-compiled binaries (called "*bottle*"), it's entirely your responsibility to get the binary both code-signed and notarized in your build system. Otherwise, should the product is accepted by the Homebrew Core team, the package is published. For GUI and etc, AutomataCI highly recommends to split it into a separate package since not everyone is using GUI interface.

9.10.4.6.2. Supported Platforms

The current AutomataCI supported host platforms are shown in Table 36.

Table 36: Homebrew packaging support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	Yes	Yes	Yes
linux-amd64 (Red Hat - Fedora)	Yes	No	Yes
darwin-amd64 (Intel-based MacOS)	Yes	Yes	Yes
windows-amd64	Yes	Yes	Yes

9.10.4.6.3. Content Assembling Function

The content assembling function is shown in Table 37.

Table 37: the content assembly function symbol for Homebrew packager

```
# POSIX Shell
PACKAGE_Assemble_HOMEBREW_Content() {
        _target="$1"
        _directory="$2"
        _target_name="$3"
        _target_os="$4"
        _target_arch="$5"

        ...
}
```

```
# PowerShell
PACKAGE-Assemble-HOMEBREW-Content {
        param(
                [string]$_target,
                [string]$_directory,
                [string]$_target_name,
                [string]$_target_os,
                [string]$_target_arch
        )

        ...
}
```

The objectives are:

1. to assemble a build-able source codes consumable by Homebrew and its build tools inside "*${_directory}*" directory; AND
2. to script the *${_directory}/formula.rb* Homebrew formula script used as a foundation for creating the finalized Homebrew formula Ruby source code.

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.10.4.6.4. Required Files

This packager has the following required files:

1. **${_directory}/formula.rb** – the Homebrew package's Ruby script.

To overcome the Homebrew "chicken and egg" problem" (see later testing package section), the *formula.rb* can have the following placeholders (exactly as it is) for value replacement during the packing process:

1. **{{ TARGET_PACKAGE }}** – the *.tar.xz* package name for AutomataCI to replace in; AND
2. **{{ TARGET_SHASUM }}** – the required *SHA256* value of the *.tar.xz* package.

Otherwise, its format complies strictly to Homebrew's Formula Cookbook (https://docs.brew.sh/Formula-Cookbook).

Should the required files are missing, AutomataCI shall fail the entire Package CI Job run.

9.10.4.6.5. Testing Package

There is literally no way to test the Homebrew package due to its "Chicken and Egg" problem it caused:

- In order to test the Homebrew formula, the Homebrew package (both *formula.rb* and *.tar.xz* payload) MUST be released and officiated first; BUT
- To create the releasable package, the package has to be tested first but it can't because the *.tar.gz* is not available in the official distribution channels.

Hence, the only way to do it is to manually test it out after release and then version control everything accordingly. Good luck!

9.10.4.7. Itsy/Open Package Packages (IPK | *OPK*)

AutomataCI supports IPK packages by default using its own packager based on Debian specification *except using **tar** instead of ar for its outer archiving layer*. This approach was required mainly because IPK is primarily used in a resources constrained environment (e.g. embedded) where GPU *ar* archiving is unavailable.

Since IPK is the predecessor to Debian *.deb* package and only being deployed in resources constrained environment, the focus is also on reducing the IPK package size to the extreme.

Although AutomataCI uses its own compiler, the output shall always be compliant with upstream. you're still required to learn through the specifications (at least binary package) shown above before proceeding to construct the job recipe. The supporting documentations AutomataCI based on are as follows:

1. https://www.debian.org/doc/debian-policy/index.html
2. https://manpages.debian.org/unstable/dpkg-dev/deb.5.en.html
3. https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html
4. https://raymii.org/s/tutorials/Building_IPK_packages_by_hand.html
5. https://downloads.openwrt.org/
6. https://nilrt-docs.ni.com/opkg/opkg_intro.html
7. https://yairgadelov.me/custom-opkg-repository/
8. https://git.yoctoproject.org/opkg/

Windows OS is not supported mainly due to the lack of UNIX permission manipulation (e.g. *chmod* or *chown* commands).

Starting from version 2.0.0, AutomataCI package IPK type in parallel execution.

9.10.4.7.1. Supported Platforms

The current AutomataCI supported host platforms are shown in Table 38.

Table 38: IPK packaging support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	**Yes**	**Yes**	**Yes**
linux-amd64 (Red Hat - Fedora)	**Yes**	**No**	**Yes**
darwin-amd64 (Intel-based MacOS)	**Yes**	**Yes**	**Yes**
windows-amd64	**No**	**No**	**N/A**

NOTE:

1. "**N/A**" – means "Not Available".

9.10.4.7.2. Content Assembling Function

The content assembling function is shown in Table 39.

Table 39: the content assembly function symbol for IPK packager

```
# POSIX Shell
PACKAGE_Assemble_IPK_Content() {
        _target="$1"
        _directory="$2"
        _target_name="$3"
        _target_os="$4"
        _target_arch="$5"

        ...
}
```

```
# PowerShell
PACKAGE-Assemble-IPK-Content {
        param(
                [string]$_target,
                [string]$_directory,
                [string]$_target_name,
                [string]$_target_os,
                [string]$_target_arch
        )

        ...
}
```

The *$_directory* variable should point to the workspace directory containing 2 important directories:

1. **control/** – housing the control components of the *.deb* package; AND
2. **data/** – housing the data components of the *.deb* package.

The objectives are:

1. to assemble the "to be installed" file structure in the **data/** directory complying to Filesystem Hierarchy Standard; AND
2. to assemble any maintainer scripts (if needed) in the **control/** directory.

For example:

1. the provided *$_target* variable that is pointing to the currently detected binary executable. It is usually being copied to **${directory}/data/usr/local/bin/** directory.
    1. This means that the package shall the target program into */usr/local/bin/* when the package is installed by the customer.

AutomataCI provides sufficient utilities to create all the required files. It's entirely your duty to assemble all the files in their respective location and only create the **control/control** file as the last step due to its requirement of calculating **data/** directory disk space consumption.

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.10.4.7.3. Required Files

This packager has the following required files:

1. *control/control*

These files follow strict format and content as specified in the Debian manual (especially *control/control* file).

9.10.4.7.4. Maintainers' Scripts

As frown by the Debian specification when used for installing files and directories, maintainers' scripts are only used when required. Manual efforts can cause unwanted and unclean executions at the end-user sides.

These optional scripts are usually used for emergency and adaptive patching, services (e.g. systemd, cron, nginx, etc) signaling and control before or after an install or uninstall process.

Supported Maintainers' scripts are:

1. **control/preinst** – commands before install
2. **control/postinst** – commands after uninstall
3. **control/prerm** – commands before uninstall
4. **control/postrm** – commands after uninstall

When in doubt, use *post[ACTION]* instead.

9.10.4.7.5. AutomataCI's Long Description Generative Function

Due to the long description length ambiguity, AutomataCI relies on external data file for generating the IPK's control file's *Description:* long data field. The file is located at:

```
${PROJECT_PATH_RESOURCES}/docs/ABSTRACTS.txt
```

Please keep in mind that this data file is also shared by other packagers or release processes for maintaining consistencies.

The file **MUST** comply to:

1. Max column length of 69 characters per line; AND
2. Strictly text-only; AND
3. UTF-8 but do only use ASCII characters for maximum backward compatibility (means English only).

9.10.4.7.6. Testing Package

Since IPK package is a multi-layer *tar* archived file, to test it, simply un-archive it and verify the file system.

9.10.4.8. MSI Packages (*Microsoft Windows Offline Installer*)

AutomataCI supports MSI packages via the use of *Wix4* and *msitools* tools depending on host machine's OS. Both yields different types of MSI installer where *msitools* could not create an UI equipped version. MSI packages is triggered when "**-msi**" keyword in its name (e.g. "myproduct-**msi**_any-any") signaling placeholder file is detected

The supporting documentations AutomataCI based on are as follows:

1. https://wiki.gnome.org/msitools
2. https://wiki.gnome.org/msitools/HowTo/CreateMSI
3. https://gitlab.gnome.org/GNOME/msitools/-/issues/2
4. https://learn.microsoft.com/en-us/previous-versions//aa372057(v=vs.85)
5. https://wixtoolset.org/docs/tools/wixext/wixui/
6. https://wixtoolset.org/docs/schema/ui/wixui/
7. https://helgeklein.com/blog/real-world-example-wix-msi-application-installer/
8. https://github.com/orgs/wixtoolset/discussions/7636
9. https://github.com/orgs/wixtoolset/discussions/7893

Starting from version 2.0.0, AutomataCI package MSI in 2 stages:

1. **LAYER 1 in series** – for generating all the required available *.wxs* manifest files when instructed by the Package CI job; AND
2. **LAYER 2 in parallel** – while still in Layer 1 execution scope, package the actual *.msi* file based on available *.wxs* package.

9.10.4.8.1. Supported Platforms

The current AutomataCI supported host platforms are shown in Table 40.

Table 40: MSI packaging support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	Yes	No	Yes
linux-amd64 (Red Hat - Fedora)	Yes	No	Yes
darwin-amd64 (Intel-based MacOS)	Yes	No	Yes
windows-amd64	Yes	No	Yes

NOTE:

1. "**N/A**" – means "Not Available".

IMPORTANT NOTICE

Due to GitHub Actions unknown bug for Windows OS (referring: https://github.com/actions/runner-images/issues/7320), there is no way we can proceed to test all the MSI processes and executions when each build takes >1hr for simple hello world build. Hence, the development is marked done but untested for now.

9.10.4.8.2. Content Assembling Function

The content assembling function is shown in Table 41.

Table 41: the content assembly function symbol for MSI packager

```
# POSIX Shell
PACKAGE_Assemble_MSI_Content() {
        _target="$1"
        _directory="$2"
        _target_name="$3"
        _target_os="$4"
        _target_arch="$5"

        ...
}
```

```
# PowerShell
PACKAGE-Assemble-MSI-Content {
        param(
                [string]$_target,
                [string]$_directory,
                [string]$_target_name,
                [string]$_target_os,
                [string]$_target_arch
        )

        ...
}
```

The objectives are:

1. to assemble all the “to be installed” files for ALL I18N languages into the *$_directory* directory; AND
2. to generate the required *.wxs* XML instruction file in its restricted filename pattern.

AutomataCI detects all files with *.wxs* file extension in Layer 2 and execute the packaging accordingly. However, due to Wix4 complexities and strict requirement, the *.wxs* filename MUST comply to the following strict pattern:

${PROJECT_SKU}_${LANG}_windows-${ARCH}.wxs

Wix4 demands culture (as in language) specification so language code must be included for parsing during parallel execution.

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.10.4.8.3. Required Files

This packager has the following required files:

1. *${_directory}/*.wxs*

These files follow strict format and content as specified in the *Wix4* and *msitools* specifications:

1. https://wixtoolset.org/docs/intro/
2. https://helgeklein.com/blog/real-world-example-wix-msi-application-installer/

9.10.4.8.4. Upstream Drama

At this point in time, compiling MSI packages for Windows-only platform can be extremely annoying and daunting due to upstream has dramas:

1. GNOME.org does not have sufficient resources and thus refused to develop the full version itself (refer: https://gitlab.gnome.org/GNOME/msitools/-/issues/) due to resources constraint; AND
2. Wix team is very pessimistic about porting its toolkit to non-Windows environment (refer: https://github.com/wixtoolset/issues/issues/4381/); AND
3. Microsoft tries to re-invent MSIX but is unable to distribute drivers like MSI (https://github.com/Microsoft/msix-packaging) package does; AND
4. WinGet team instead of solving the MSI complication problem, forces developers to go through it instead before up-streaming to them (https://github.com/microsoft/wingetpkgs).

Until Microsoft resolve their office-politic dramas, we as external developers have to go through them without a choice. Please do not get in-between their office politics.

9.10.4.8.5. I18N Supports (Multi-lingual)

By default and by design, MSI package is a single language, first impression user experience. Hence, to support multiple languages, each language will be compiled into its own package. This means multiple *.wxs* file with language code specified are created independently.

Take note that in Microsoft Windows, they use the LCID number (refer: https://learn.microsoft.com/en-us/previous-versions/windows/embedded/ms912047(v=winembedded.10)). Hence, AutomataCI automatically queries the LCID number from the given language code parsed from the filename.

In the *.wxs* scripting during content assembly function execution, one must set the LCID number accordingly in the designated variable. Table 42 shows some examples.

Table 42: Some examples of LCID number settings while scripting *.wxs* XML file

Language	**Code**	**Definitions** **(top: *POSIX Shell*; bottom: *PowerShell*)**
Simplified Chinese	*zh-hans*	__var_LANGUAGE_ID='2052' $__var_LANGUAGE_ID = ‘2052’
English United States	*en-us*	__var_LANGUAGE_ID='1033' $__var_LANGUAGE_ID = '1033'
Swedish Sweden	*sv-se*	__var_LANGUAGE_ID='1053' $__var_LANGUAGE_ID = '1053'
German from Germany	*de-de*	__var_LANGUAGE_ID='1031' $__var_LANGUAGE_ID = '1031'
Low German (Germany)	*nds-de*	__var_LANGUAGE_ID='4096' $__var_LANGUAGE_ID = '4096'

9.10.4.8.6. Graphical User Interface (GUI) Support

MSI GUI installer support is only available on Windows OS packager (via *Wix4*). Due to Wix4 .NET extensions complexities, AutomataCI sorts out its dependencies internally and autonomously (including download and extract) in order to manage them seamlessly without sacrificing customization.

The GUI specifications are referenced from:

1. https://wixtoolset.org/docs/tools/wixext/
2. https://wixtoolset.org/docs/schema/ui/wixui/
3. https://wixtoolset.org/docs/tools/wixext/wixui
4. https://github.com/orgs/wixtoolset/discussions/6623

GUI is not available on UNIX and MacOS built MSI packages as *msitools* does not support it at the moment.

9.10.4.8.7. Banners and Icons Customization

By default, the banners and icons can be customized complying to Wix4 requirements (refer: https://wixtoolset.org/docs/tools/wixext/wixui/#replacing-the-default-bitmaps). This is not applicable for *msitools* (Linux & MacOS build). Table 43 listed the specifications for the assets creations.

Table 43: Recommended UI assets customization settings for MSI Packager

Type	**Format**	**Width**	**Height**	**Resolutions**
Banner (*WixUIBannerBmp*)	JPEG	493	58	90dpi
Banner (*WixUIDialogBmp*)	JPEG	493	312	90dpi
ICO Icon (<*Icon*>)	ICO	(Layer 1) 16 (Layer 2) 32 (Layer 3) 48 (Layer 4) 64 (Layer 5) 128 (Layer 6) 256	(Layer 1) 16 (Layer 2) 32 (Layer 3) 48 (Layer 4) 64 (Layer 5) 128 (Layer 6) 256	32bpp 8bit alpha

9.10.4.8.8. GUID Produce & Upgrade Code

MSI Packages heavily used many *GUID* (or *UUID*) random codes to identify many items such as but not limited to:

1. **Product Code** – variable *UUID* for unique identification.
2. **Upgrade Code** – persistent *UUID* software identification.
3. **Components GUID** – persistent *UUID* for components identifications.

Each i18N language should posses different *UUID* codes as well except *Upgrade Code* where it should be persistent anywhere everywhere.

9.10.4.8.9. Testing Package

There is no direct way of testing the package except setting up a virtual machine with Windows OS and attempting to install it manually.

9.10.4.9. Open Container (Docker | Podman)

AutomataCI supports Open Container primarily using Docker to build a cross-platform compatible when an executable is detected. The container image is horizontally scalable, and automated orchestration capable by default. The supporting documentations AutomataCI based on are as follows:

1. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
2. https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html
3. https://medium.com/@kelseyhightower/optimizing-docker-images-for-static-binaries-b5696e26eb07
4. https://docs.docker.com/build/building/multi-platform/#building-multi-platform-images
5. https://docs.docker.com/build/building/base-images/#create-a-simple-parent-image-using-scratch
6. https://docs.docker.com/engine/reference/builder/
7. https://docs.docker.com/build/attestations/slsa-provenance/
8. https://github.com/orgs/community/discussions/45969
9. https://github.com/opencontainers/image-spec/blob/main/annotations.md#pre-defined-annotation-keys
10. https://docs.github.com/en/packages/learn-github-packages/connecting-a-repository-to-a-package
11. https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry
12. https://www.docker.com/blog/multi-arch-build-and-images-the-simple-way/
13. https://docs.github.com/en/packages/managing-github-packages-using-github-actions-workflows/publishing-and-installing-a-package-with-github-actions

Due to the container registry synchronization limitations, AutomataCI can only package Open Container type in serial execution.

9.10.4.9.1. Supported Platforms

The current AutomataCI supported host platforms are shown in Table 44.

Table 44: Open Container packaging support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	Yes	Yes	Yes
linux-amd64 (Red Hat - Fedora)	Yes	No	Yes
darwin-amd64 (Intel-based MacOS)	No	No	No
windows-amd64	Yes	No	Yes

9.10.4.9.2. Content Assembling Function

The content assembling function is shown in Table 45.

Table 45: the content assembly function symbol for Open Container packager

```
# POSIX Shell
PACKAGE_Assemble_CONTAINER_Content() {
        _target="$1"
        _directory="$2"
        _target_name="$3"
        _target_os="$4"
        _target_arch="$5"

        ...
}
```

```
# PowerShell
PACKAGE-Assemble-CONTAINER-Content {
        param(
                [string]$_target,
                [string]$_directory,
                [string]$_target_name,
                [string]$_target_os,
                [string]$_target_arch
        )

        ...
}
```

The objectives are:

1. to assemble all the required files and resources into this directory; AND
2. to generate the required *Dockerfile* (named as it is) specific to the target OS and CPU architecture.

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.10.4.9.3. Required Files

This packager has the following required files:

1. *Dockerfile* – the control file for building Open Container image either via Docker or Podman.

Should the *Dockerfile* is missing, the Package CI job shall fail immediately.

IMPORTANT NOTE

Although the market now has *Podman*, a competitor to *Docker*, AutomataCI will use *Docker* by default. The control file shall retain *Dockerfile* until further notice.

9.10.4.9.4. Docker Base Images

To keep things minimal, the recommended (not a rule) base images are tabulated in Table 46. AutomataCI encourages small Container Image (<GB) and full static execution.

Table 46: AutomataCI recommended base container image

Runtime OS	**App Type**	**Recommendation**
Linux	Pure Static	--platform=${_target_os}/${_target_arch} scratch
Linux	Dynamic	--platform=${_target_os}/${_target_arch} linuxcontainers/debian-slim:latest
Unknown	Pure Static	--platform=${_target_os}/${_target_arch} scratch
Unknown	Dynamic	--platform=${_target_os}/${_target_arch} linuxcontainers/debian-slim:latest
Windows	Pure Static, Dynamic	--platform=${_target_os}/${_target_arch} mcr.microsoft.com/windows/nanoserver:ltsc2022
Darwin	Pure Static	Not supported
Darwin	Dynamic	Not supported

To workaround of creating some required directory in the *scratch* type image, simply script a empty *.tmpfile* to the destination directory and copy into it. Example, to create */tmp* directory, the following instruction is used (assuming the .blank empty file is created):

```
COPY .blank /tmp/.tmpfile
```

9.10.4.9.5. Registry Synchronization

In order to build multi-platform images (not just Linux but Windows), image registry (e.g. GitHub Packages) synchronization is a mandatory requirement. Hence, while performing an image build, **AutomataCI also releases the image at the same time**.

9.10.4.9.6. Open Container Initiative (OCI) Compatibility

Although AutomataCI can build Docker-specific image, to ensure the built images are available to as many containers' ecosystem as possible, AutomataCI heavily complies to OCI's engineering specifications. This means:

1. Docker-specific metadata are removed.
    1. build with *BUILDX_NO_DEFAULT_ATTESTATIONS=1* environment variable and *--provenance=false, --sbom=false* arguments.
2. The label "*org.opencontainers.image.ref.name*" is automatically filled in the build command via the argument
    1. --label "*org.opencontainers.image.ref.name=${__tag}*"

Hence, manual filling in the *Dockerfile* is not required.

9.10.4.9.7. Required Secret Data

As stated earlier, AutomataCI requires a number of secret environment variables in order to operate properly:

1. **PROJECT_CONTAINER_REGISTRY** (*CONFIG.toml*) – defines the registry's handle.
2. **PROJECT_SOURCE_URL** (*CONFIG.toml*) – Defines the source code location.
3. **CONTAINER_USERNAME** (*SECRETS.toml*) – Use for registry login account identification.
4. **CONTAINER_PASSWORD** (*SECRETS.toml*) – Use for registry login authentication.

9.10.4.9.8. Testing Package

To test a built open container package, you can only run locally by pulling it from the registry manually using run action (e.g. *docker run* or *podman run*).

9.10.4.10. PyPi Packages (Python *Programming Language*)

AutomataCI supports PyPi packages for Python Programming Language native ecosystem when a signaling placeholder file with "**-src**" keyword in its name (e.g. "myproduct**-src**_any-any") and active **$PROJECT_PYTHON** are simultaneously detected. The supporting documentations AutomataCI based on are as follows:

1. https://packaging.python.org/en/latest/specifications/
2. https://packaging.python.org/en/latest/specifications/declaring-project-metadata/#declaring-project-metadata
3. https://peps.python.org/pep-0660/
4. https://pypi.org/project/twine/

AutomataCI employs the clean-slate library assembling (similar to first time upload to PyPi) for consistency assurances and for providing maximum freedom to developers in the case of library-app repository use.

Starting from version 2.0.0, AutomataCI package PyPi type in parallel execution.

9.10.4.10.1. Supported Platforms

The current AutomataCI supported host platforms are shown in Table 47.

Table 47: PyPi packaging support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	**Yes**	**Yes**	**Yes**
linux-amd64 (Red Hat - Fedora)	**Yes**	**Yes**	**Yes**
darwin-amd64 (Intel-based MacOS)	**Yes**	**Yes**	**Yes**
windows-amd64	**Yes**	**Yes**	**Yes**

9.10.4.10.2. Content Assembling Function

The content assembling function is shown in Table 48.

Table 48: the content assembly function symbol for PyPi packager

```
# POSIX Shell
PACKAGE_Assemble_PYPI_Content() {
        _target="$1"
        _directory="$2"
        _target_name="$3"
        _target_os="$4"
        _target_arch="$5"

        ...
}
```

```
# PowerShell
PACKAGE-Assemble-PYPI-Content {
        param(
                [string]$_target,
                [string]$_directory,
                [string]$_target_name,
                [string]$_target_os,
                [string]$_target_arch
        )

        ...
}
```

The objectives are:

1. to assemble the Python library “as it is” into *$_directory* directory; AND
2. optionally, to script *pyproject.toml* instruction file.

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.10.4.10.3. Required Files

This packager has the following required files:

1. *pyproject.toml* – instructions for Python to package PyPi package.
2. ***README** file (defined in **$PROJECT_PYPI_README**)* – required intro file.

Should any of the required file is missing, AutomataCI shall throw an error and stop the packaging job entirely.

Should *pyproject.toml* file is missing, AutomataCI shall generate a default file on-behalf to fulfill the construction requirement. Due to its complexities, you are strong encouraged to generate the file during the content assembling function phase to match your actual Project requirement.

9.10.4.10.4. Dependencies

AutomataCI uses *twine* Python PyPi package. Hence, please include it inside the *requirements.txt* in the project.

9.10.4.10.5. Testing Package

twine pip module can be used for testing the generated packages using the following command:

```
$ twine check "${__directory}/dist/"*
Checking ./AutomataCI-1.5.0-py3-none-any.whl: PASSED
Checking ./AutomataCI-1.5.0.tar.gz: PASSED
```

9.10.4.11. RPM Packages (Red Hat *OS*)

AutomataCI supports RPM packages using *rpmbuild* toolkit supplied by Red Hat. The supporting documentations AutomataCI based on are as follows:

1. https://rpm-software-management.github.io/rpm/manual/spec.html
2. http://ftp.rpm.org/api/4.4.2.2/specfile.html
3. https://developers.redhat.com/blog/2019/03/18/rpm-packaging-guide-creating-rpm
4. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html-single/rpm_packaging_guide/index#rpm-packages_packaging-software
5. https://stackoverflow.com/questions/15055841/how-to-create-spec-file-rpm
6. https://stackoverflow.com/questions/27862771/how-to-produce-platform-specific-and-platform-independent-rpm-subpackages-from-o
7. https://unix.stackexchange.com/questions/553169/rpmbuild-isnt-using-the-current-working-directory-instead-using-users-home

AutomataCI prioritizes binary package build since the source codes are usually distributed using git control ecosystem. This removes any duplication related to the project and focus on customer delivery instead.

This binary package is a Red-Hat based Linux OS (e.g. Fedora, CentOS, etc) exclusive package so Windows and MacOS OSes are naturally by default: not supported.

Starting from version 2.0.0, AutomataCI package RPM type in parallel execution.

9.10.4.11.1. Supported Platforms

The current AutomataCI supported host platforms are shown in Table 49.

Table 49: RPM packaging support status

Platform	Supported	Tested	Automated Test
linux-amd64 (Debian)	**Yes**	**Yes**	**Yes**
linux-amd64 (Red Hat - Fedora)	**Yes**	**No**	**Yes**
darwin-amd64 (Intel-based MacOS)	**N/A**	**N/A**	**N/A**
windows-amd64	**N/A**	**N/A**	**N/A**

NOTE:

1. "**N/A**" – means "Not Available".

9.10.4.11.2. Content Assembling Function

The content assembling function is shown in Table 50.

Table 50: the content assembly function symbol for RPM packager

```
# POSIX Shell
PACKAGE_Assemble_RPM_Content() {
        _target="$1"
        _directory="$2"
        _target_name="$3"
        _target_os="$4"
        _target_arch="$5"

        ...
}
```

```
# PowerShell
PACKAGE-Assemble-RPM-Content {
        param(
                [string]$_target,
                [string]$_directory,
                [string]$_target_name,
                [string]$_target_os,
                [string]$_target_arch
        )

        return 10 # not supported in Windows OS
}
```

The *$_directory* variable should point to the workspace directory containing 2 important directories:

1. **BUILD/** – housing the data components of the *.rpm* package; AND
2. **SPECS/** – housing the control components of the *.rpm* package.

The objectives are:

1. to assemble the “to be installed” file structure in the **${_directory}/BUILD/** directory; AND
2. to assemble all SPEC files fragments in the **${_directory}/SPECS/** directory.

For example:

1. The provided *$_target* variable that is pointing to the currently detected binary executable. It is usually being copied to **${directory}/BUILD/** directory.
2. Then spin the “install” segment spec fragment file into **${directory}/SPECS** as shown in Table 51.

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.10.4.11.3. Required Files

This packager has the following required files:

1. ${_directory}/SPEC_INSTALL
2. ${_directory}/SPEC_FILES

These files follow strict format and content as specified in the RPM specifications listed below:

1. https://rpm-software-management.github.io/rpm/manual/spec.html
2. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html-single/rpm_packaging_guide/index#an-example-spec-file-for-bello_working-with-spec-files

The content of the *${_directory}/SPEC_INSTALL* file is the ***%install*** commands that will be copied over during AutomataCI's automatic spec file generation. You're required to provide the instructions (without the ***%install*** stanza) in the content assembly function. An example command is shown in Table 51.

Table 51: an example of "INSTALL" segment of spec file command

```
# generate AutomataCI's required RPM spec instructions (INSTALL)
printf -- "\
install --directory %%{buildroot}/usr/local/bin
install -m 0755 ${PROJECT_SKU} %%{buildroot}/usr/local/bin

install --directory %%{buildroot}/usr/local/share/doc/${PROJECT_SKU}/
install -m 0644 copyright %%{buildroot}/usr/local/share/doc/${PROJECT_SKU}/

install --directory %%{buildroot}/usr/local/share/man/man1/
install -m 0644 ${PROJECT_SKU}.1.gz %%{buildroot}/usr/local/share/man/man1/
" >> "${__directory}/SPEC_INSTALL"
```

The content of the *${_directory}/SPEC_FILES* file is the ***%files*** commands that will be copied over during AutomataCI's automatic spec file generation. You're required to provide the instructions (without the ***%files*** stanza) in the content assembly function

An example command is shown in Table 52.

Table 52: an example of "FILES" segment of spec file command

```
# generate AutomataCI's required RPM spec instructions (INSTALL)
printf -- "\
/usr/local/bin/${PROJECT_SKU}
/usr/local/share/doc/${PROJECT_SKU}/copyright
/usr/local/share/man/man1/${PROJECT_SKU}.1.gz
" >> "${__directory}/SPEC_FILES"
```

9.10.4.11.4. Optional Specs Files

AutomataCI also provides other Spec's Fragment Files for overriding specific fields in the spec file generation such as but not limited to:

1. *${_directory}/SPEC_DESCRIPTION*
2. *${_directory}/SPEC_PREPARE*
3. *${_directory}/SPEC_BUILD*
4. *${_directory}/SPEC_CLEAN*
5. *${_directory}/SPEC_CHANGELOG*

All specifications are available at:

1. https://rpm-software-management.github.io/rpm/manual/spec.html
2. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html-single/rpm_packaging_guide/index#an-example-spec-file-for-bello_working-with-spec-files

If the *${_directory}/SPEC_DESCRIPTION* spec fragment file is **not provided**, AutomataCI shall automatically parse the following data file from:

```
${PROJECT_PATH_SOURCE}/docs/ABSTRACTS.txt
```

If the *${_directory}/SPEC_CHANGELOG* file is **not provided**, AutomataCI shall automatically parse and process data from:

```
${PROJECT_PATH_SOURCE}/changelog/data/latest
```

GENTLE REMINDER

Likewise, stanza (e.g. **%description**) is not required. Only the content is permitted to be in the file.

9.10.4.11.5. Overrides Entire Spec File Manually

To override the entire spec file for full manual control, simply create a fully compliant spec file in the content assembly function at the following location:

```
${_directory}/SPECS/${PROJECT_SKU}.spec
```

Should AutomataCI detects the existence of such file, the generative function is skipped entirely.

9.10.4.11.6. License SPDX Data

RPM requires an explicit declaration of the project's license's SPDX ID. To ensure consistencies across all package ecosystem, AutomataCI uses *$PROJECT_LICENSE* value directly. Please change the value from there accordingly in *CONFIG.toml* file. Known SPDX IDs are available at: https://spdx.org/licenses/.

9.10.4.11.7. Canceling Default Built-In Definitions

To ensure the packaging does not perform any alterations to the end product or against the host OS, AutomataCI deploys the following definitions by default to prevent *rpmbuild* from performing magical changes:

```
$ rpmbuild \
        --define "_topdir ${__directory}" \ # only work in workspace; not elsewhere
        --define "debug_package %{nil}" \ # ensure not to alter any packaged product.
        --define "__strip /bin/true" \ # ensure not to alter any packaged product.
        --target "$__arch" \
        -ba "$__.spec"
```

9.10.4.11.8. Testing Package

To test a built *.rpm* package, simply use the following commands (when available):

```
$ rpm -K [Name].rpm
[Name].rpm: digests OK
```

9.11. Release

Release CI job operates by scanning through the *$PROJECT_PATH_ROOT/$PROJECT_PATH_PKG/* directory for the all subjects. It performs the required checking, finalization, generate the necessary release assets files, and finally upstream to the distributions ecosystem.

IMPORTANT NOTE

This is, by default, a forward moving milestone CI Job usually triggered manually. The automated processes are mainly to validate the Release CI Job is working fine. ***Some executions however, are irreversible (e.g. packages are already published)***. To reverse the outcome of this job, depending on severity, you might have to use the version control system to roll back, clean up the workspace, and depending on the remote ecosystem availability, override or remove the published packages.

Therefore, **be absolutely ready before calling this CI job**.

9.11.1. Operating Mechanism

This CI Job executes in series by default due to finalization process. Parallel execution is strongly not recommended.

It's highly recommended to keep all the release CI job algorithms inside the baseline job recipe only.

9.11.2. Cryptography Signing

It's duly noted that some ecosystems require cryptography notorization such as but not limited to GPG signing for *.deb* and *.rpm* package types.

9.11.3. Limited Customization

Release CI job offers limited CI job recipe customization for only baseline (*$PROJECT_PATH_SOURCE*) source. AutomataCI shall source their *.ci/release_unix-any.sh* and *.ci/release_windows-any.ps1* job recipes.

The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources ***release_unix-any.sh*** or ***release_windows-any.ps1*** (represented as "***release_{unix,windows}-any.{sh,ps1}***") automataCI execution.
4. ***release_{unix,windows}-any.{sh,ps1}*** sources common content assembling function from baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/release_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/release_windows-any.ps1*.**
5. ***release_{unix,windows}-any.{sh,ps1}*** runs the supplied pre-release execution function for customized executions before performing scans.
6. ***release_{unix,windows}-any.{sh,ps1}*** scans each packages and execute finalization. Upon completing its default functions, it calls the supplied in-flight execution function while feeding the subject for custom operation.
7. ***release_{unix,windows}-any.{sh,ps1}*** performs the sealing of the release including modification to repository's version control components.
8. ***release_{unix,windows}-any.{sh,ps1}*** runs the supplied post-release execution function for customized executions before publications.
9. ***release_{unix,windows}-any.{sh,ps1}*** runs the publications and upstream function.
10. ***release_{unix,windows}-any.{sh,ps1}*** concludes the operation.
11. Maintainer can initialize the next version and commit everything as milestone.

9.11.3.1. Supplicant Functions

There are 3 optional supplicant functions depending on the state of Release CI Job executions. Usually, they are not needed and left empty. They are:

1. **RELAESE_Run_Pre_Processing** – last overall catch-up before executing finalization.
2. **RELEASE_Run_Package_Processing** – last individual package executions.
3. **RELEASE_Run_Post_Processing** – last touch up before publishing the packages.

The following return numbers to tell AutomataCI to perform the necessary actions:

(a) **10** – Tell AutomataCI to skip the packaging process.

(b) **0** – All good and proceed.

(c) **non-0** – Error is found.

9.11.4. Upstream Repository Location

Depending on where the publication repository located, AutomataCI shall pull and setup them locally in order to publish the applicable packages. Any remote publication repository in the local environment are located in:

```
$PROJECT_PATH_ROOT/$PROJECT_PATH_RELEASE/
```

9.11.5. Affected Packages

Different packages are acted differently by Release CI Job. They are detailed in their sub-sections.

Unlike other CI jobs, should any of the technology or package type is missing in the sub-sections, it is safely assumed that Release CI Job does nothing onto the subjected package to keep this book effectively thinner.

9.11.5.1. Apt Repository (.deb Package)

AutomataCI publish an *.deb* package by using *reprepro* to setup/update existing APT repository system upon detecting:

1. a file with a "*.deb*" file extension.

Example: "*automataci_1.7.0_linux-arm64**.deb***"

Since APT is a static filesystem, AutomataCI use *git* technology to clone a given ***$PROJECT_STATIC_REPO*** into place a copy into its ***deb/*** directory. The full local directory path is as follows:

```
${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}/${PROJECT_STATIC_REPO_DIRECTORY}/deb/

WHERE $PROJECT_STATIC_REPO is:
${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}/${PROJECT_STATIC_REPO_DIRECTORY}/
```

AutomataCI assumes the host system has an already authenticated access to the ***$PROJECT_STATIC_REPO*** set in *CONFIG.toml* (e.g. configured *SSH*).

Upon successful publication, the local package directory shall be kept in case of upstreaming to another ecosystem. However, **this is an irreversible 1-way execution**.

AutomataCI relies on all the environment variables tabulated in Table 53 to login and upstream the package.

Table 53: Required environment variables for AutomataCI to upstream DEB.

Name	**Supplied By**	**Purpose**
PROJECT_STATIC_REPO	*CONFIG.toml*	The git repository URL to setup locally.
PROJECT_STATIC_REPO_KEY	*CONFIG.toml*	The git upstream key like "***origin***".
PROJECT_STATIC_REPO_BRANCH	*CONFIG.toml*	The git branch name to work in.
PROJECT_STATIC_REPO_DIRECTORY	*CONFIG.toml*	The directory name for housing the git repository.

9.11.5.1.1. Only Works in UNIX OS

Due to *reprepro* availability and its nature, this release **only works on UNIX OS (Linux and MacOS) only**.

9.11.5.1.2. Reprepro Distribution Config File

Reprepro requires a *conf/distributions* configuration file to operate on. AutomataCI facilitates it by automatically generating one based on its dataset and stored it in:

```
${PROJECT_PATH_ROOT}/${PROJECT_PATH_TEMP}/deb/
```

This file is manually analyzable and shall only being used to assemble all the latest .deb packages in an apt repository.

9.11.5.1.3. Supported Architectures

Due to the designed factor of *reprepro*, AutomataCI has to use a combinations list of OSes and CPU architectures listed from "*dpkgarchitectures -L*" command (e.g. amd64 CPU architecture with linux and openbsd OSes yield "*amd64 linux-amd64 openbsd-amd64*").

These OS and CPU Architecture values are updated from time to time based on Debian stable OS until a suitable approach is found. The current lists are available in *automataCI/services/publishers/reprepro.[EXTENSION]* library.

9.11.5.1.4. Reprepro Database

Due to the fact that one may keep the database for future referencing, AutomataCI retains its database just in case in the following directory:

```
${PROJECT_PATH_ROOT}/${PROJECT_PATH_TEMP}/deb/db
```

If there are later version being released, the database should be working fine. Otherwise, one has to delete the database and reassemble the packages from scratch again.

9.11.5.2. Cargo Registry (*Rust Programming Language*)

AutomataCI automatically login into a Cargo registry and publish the cargo package upon detecting:

1. a directory with a name containing "**cargo**_" prefix (e.g. "cargo_myproduct -cargo_any-any"); AND
2. a valid *Cargo.toml* inside the directory.

Upon successful publication, the local package directory shall be deleted by AutomataCI automatically. Hence, **this is an irreversible 1-way execution**.

AutomataCI relies on all the environment variables tabulated in Table 54 to login and upstream the package.

Table 54: Required environment variables for AutomataCI to upstream Cargo.

Name	**Supplied By**	**Purpose**
CARGO_REGISTRY	*CONFIG.toml*	Registry location. Default is '***crates-io***'.
CARGO_PASSWORD	*SECRETS.toml*	The secret password or the secret API token provided by the registry

9.11.5.3. Changelog

AutomataCI automatically seals the "*latest*" changelog entry found in the following directory into *$PROJECT_VERSION* filename:

```
$PROJECT_PATH_ROOT/$PROJECT_PATH_SOURCE/changelog/{data,deb}/
```

Upon successful publication, the "latest" entry will be missing and a new version numbered entry appears. Once done, some CI Job shall fail until a new "*latest*" entry is created with a newer *$PROJECT_VERSION* value.

To reverse this changes, you can manually rename the *$PROJECT_VERSION* entry back to "latest" in their respective directory. Hence, **this is a reversible 2-way execution**.

9.11.5.4. Chocolatey Repository

AutomataCI automatically login into a Cargo registry and publish the cargo package upon detecting:

1. a zip package with a name containing "**-chocolatey**" keyword; AND
2. a "*.nupkg*" file extension.

Example: "myproduct**-chocolatey**_any-any**.nupkg**"

Since Chocolatey does not have any authentication management system elsewhere aside their core ecosystem, AutomataCI use *git* technology to clone a given **$PROJECT_CHOCOLATEY_REPO** into place a copy into its **Packages/** directory.

AutomataCI assumes the host system has an already authenticated access to the **$PROJECT_CHOCOLATEY_REPO** set in *CONFIG.toml* (e.g. configured *SSH*).

Upon successful publication, the local package directory shall be kept in case of upstreaming to another ecosystem. However, **this is an irreversible 1-way execution**.

9.11.5.5. Citation (CITATION.cff)

Starting from version v2.0.0, AutomataCI automatically generates a *CITATION.cff* metadata file for academic or journalism referencing purposes. The documentation AutomataCI based on are:

1. https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-citation-files#citation-formats
2. https://github.com/citation-file-format/ruby-cff
3. https://github.com/citation-file-format/citation-file-format/tree/main
4. https://citation-file-format.github.io/

By default, AutomataCI generates the file with the following filepath:

```
$PROJECT_PATH_ROOT/$PROJECT_PATH_PKG/${PROJECT_SKU}-CITATION_${PROJECT_VERSION}.cff
```

Example:

```
$PROJECT_PATH_ROOT/$PROJECT_PATH_PKG/myproduct-CITATION_v2.0.0.cff
```

Upon successful creation, a copy shall overwrite the repository's *CITATION.cff* to indicates its successful update.

To reverse this changes, you have to use your version control system to revert CITATION.cff back to its initial version. Hence, **this is a reversible 2-way execution**.

9.11.5.5.1. Abstract Customization

AutomataCI relies on an external dependency to construct the *abstract* segment in the *CITATION.cff* file located in:

```
$PROJECT_PATH_ROOT/$PROJECT_PATH_SOURCE/docs/ABSTRACTS.txt
```

This same file is also used elsewhere for maintaining consistency across many ecosystems.

9.11.5.5.2. Appendix Attachment

By default, AutomataCI seeks an attachable YAML data file and fuse it into *CITATION.cff* located at:

```
$PROJECT_PATH_ROOT/$PROJECT_PATH_SOURCE/docs/CITATIONS.yml
```

These fields are the segments that are incompatible with AutomataCI dataset. Hence, Anything else found in the specified file shall be appended as it is. Please keep in mind that this is still a YAML file and contains strict manifest fields. Table 55 shows the minimum fields found by AutomataCI as of version v2.0.0.

Table 55: The minimum fields to be attached as of version v2.0.0.

```
message: |-
  Please cite and reference this repository accordingly.
authors:
  - given-names: "Kean Ho"
    family-names: "Chew"
    email: "hollowaykeanho@gmail.com"
    affiliation: "Independent"
    orcid: "https://orcid.org/0000-0003-4202-4863"
  - given-names: "Cory"
    family-names: "Galyna"
    email: "124406765+corygalyna@users.noreply.github.com"
    affiliation: "Independent"
identifiers:
  - type: doi
    value: "10.5281/zenodo.000000"
    description: "Paper"
keywords:
  - "continuous integration"
  - "native and locally available"
  - ...
```

9.11.5.6. Docs Repository

AutomataCI detects the documentation directory located at:

```
${PROJECT_PATH_ROOT}/${PROJECT_PATH_DOCS}
```

Should it exists, AutomataCI shall only update and publish the content if exists.

Upon successful publication, the static website shall be updated shortly. Hence, **this is an irreversible 1-way execution**.

AutomataCI relies on all the environment variables tabulated in Table 56 to login and upstream the package.

Table 56: Required environment variables for AutomataCI to upstream Cargo.

Name	**Supplied By**	**Purpose**
PROJECT_DOCS_REPO	*CONFIG.toml*	Static website repository.
PROJECT_DOCS_REPO_KEY	*CONFIG.toml*	The git upstream key like "*origin*".
PROJECT_DOCS_REPO_BRANCH	*CONFIG.toml*	The git branch to save into.
PROJECT_DOCS_REPO_DIRECTORY	*CONFIG.toml*	The directory name housing the locally pulled repository.

9.11.5.7. Homebrew Repository

AutomataCI automatically publish (either by create or override) the Homebrew formula file fulfilling the following criteria:

1. a file with a name containing "**-homebrew**" keyword; AND
2. a "***.rb***" Ruby Programming Language extension.

Example:

automataci**-homebrew**_1.7.0_any-any**.rb**

Since Homebrew registry is essentially a git repository with an active **Formula/** directory, AutomataCI can clone the git repository and publish the formula into it, effectively upstream the formula file.

Upon successful publication, the formula file should be available downstream. Hence, **this is an irreversible 1-way execution**.

AutomataCI relies on all the environment variables tabulated in Table 57 to login and upstream the package.

Table 57: Required environment variables for AutomataCI to upstream Homebrew.

Name	**Supplied By**	**Purpose**
PROJECT_HOMEBREW_REPO	*CONFIG.toml*	The git repository url.
PROJECT_HOMEBREW_REPO_KEY	*CONFIG.toml*	The git upstream key like "*origin*".
PROJECT_HOMEBREW_REPO_BRANCH	*CONFIG.toml*	The git branch to save into.
PROJECT_HOMEBREW_REPO_DIRECTORY	*CONFIG.toml*	The directory name housing the locally pulled repository.

9.11.5.8. Open Container Registry (Docker | Podman)

AutomataCI assemble the "*latest*" and "*$PROJECT_VERSION*" tag via the Docker/Podman manifest management facility. The supported documents AutomataCI based on are:

1. https://docs.docker.com/engine/reference/commandline/manifest/

Upon the completion:

1. The "*latest*" tag has been created or updated.
2. A generic multi-arch image "*VERSION*" tag is created or updated.

Depending on registry, by default, the same version images can be overridden. Hence, this is a **2-way reversible execution**.

AutomataCI relies on all the environment variables tabulated in Table 58 to login and upstream the package.

Table 58: Required environment variables for AutomataCI to upstream Open Container.

Name	**Supplied By**	**Purpose**
PROJECT_CONTAINER_REGISTRY	*CONFIG.toml*	defines the registry's handle.
PROJECT_SOURCE_URL	*CONFIG.toml*	Defines the source code location.
CONTAINER_USERNAME	*SECRETS.toml*	Use for registry login account identification.
CONTAINER_PASSWORD	*SECRETS.toml*	Use for registry login authentication.

9.11.5.9. PyPi Registry (Python *Programming Language*)

AutomataCI automatically uses a pip-installed *twine* program to upstream a PyPi library when a directory with the following conditions is detected:

1. Filename has a prefix "**pypi-**"; AND
2. Is housing a *.whl* (zip format) archive and a *.tar.gz* archive.

Duly noted that PyPi registry usually prohibits the same version update and they shall reject it as an error attempt. Upon successful publication, the local package directory shall be deleted by AutomataCI automatically. Hence, **this is an irreversible 1-way execution**.

AutomataCI relies on all the environment variables tabulated in Table 59 to login and upstream the package.

Table 59: Required environment variables for AutomataCI to upstream PyPi.

Name	**Supplied By**	**Purpose**
PROJECT_PYPI_REPO_URL	*CONFIG.toml*	The registry URL to publish into.
TWINE_USERNAME	*CONFIG.toml*	username instructed by the package registry (usually "**__token__**").
TWINE_PASSWORD	*SECRETS.toml*	private token issued by the package registry.

9.11.5.10. RPM Repository (.rpm Package)

AutomataCI publish a *.rpm* package by using *createrepo_c* to setup/update existing RPM repository system upon detecting:

1. a file with a "*.rpm*" file extension.

Example: "*automataci-1.7.0-1.amd64**.rpm***"

Since RPM repository is a static filesystem, AutomataCI use *git* technology to clone a given ***$PROJECT_STATIC_REPO*** into place a copy into its ***rpm/*** directory. The full local directory path is as follows:

${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}/${*PROJECT_STATIC_REPO_DIRECTORY}*/rpm/ WHERE $PROJECT_STATIC_REPO is: ${PROJECT_PATH_ROOT}/${PROJECT_PATH_RELEASE}/${*PROJECT_STATIC_REPO_DIRECTORY}*/

AutomataCI assumes the host system has an already authenticated access to the ***$PROJECT_STATIC_REPO*** set in *CONFIG.toml* (e.g. configured *SSH*).

Upon successful publication, the local package directory shall be kept in case of upstreaming to another ecosystem. However, **this is an irreversible 1-way execution**.

AutomataCI relies on all the environment variables tabulated in Table 60 to login and upstream the package.

Table 60: Required environment variables for AutomataCI to upstream RPM.

Name	**Supplied By**	**Purpose**
PROJECT_STATIC_REPO	*CONFIG.toml*	The git repository URL to setup locally.
PROJECT_STATIC_REPO_KEY	*CONFIG.toml*	The git upstream key like "***origin***".
PROJECT_STATIC_REPO_BRANCH	*CONFIG.toml*	The git branch name to work in.
PROJECT_STATIC_REPO_DIRECTORY	*CONFIG.toml*	The directory name for housing the git repository.

9.11.5.10.1. Only Works in Linux

Due to the nature of *createrepo_c* program (refer: https://github.com/rpm-software-management/createrepo_c), **this only works on Linux host system only**.

9.12. Deploy

Deploy CI job operates by executing one/many production fleet(s) to roll out update with the newly released version. This is facilitated in case of any production-level services swarm management like Kubernetes (https://kubernetes.io/) roll-out across the globe.

9.12.1. Operating Mechanism

The default CI Job executes in series due to its lightweight role. Parallel executions are available.

9.12.2. Customization

Deploy CI job offers CI job recipe customization for baseline (*$PROJECT_PATH_SOURCE*) source. AutomataCI shall source its *.ci/deploy_unix-any.sh* and *.ci/deploy_windows-any.ps1* job recipes.

The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources ***common_unix-any.sh*** or ***common_windows-any.ps1*** (represented as "***common_{unix,windows}-any.{sh,ps1}***") automataCI execution.
4. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/deploy_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/deploy_windows-any.ps1*.**

9.13. Clean

Clean CI job operates by providing a customizable repository clean up execution sequences. The hardness of the CI job depends solely on how the project developers created it. By default, this CI job do nothing.

9.13.1. Operating Mechanism

The default CI Job executes in series due to its lightweight role. Parallel executions are available.

9.13.2. Customization

Start CI job offers CI job recipe customization for both baseline (*$PROJECT_PATH_SOURCE*) and tech-specific (e.g. Go for *$PROJECT_GO*) sources. AutomataCI shall source their *.ci/clean_unix-any.sh* and *.ci/clean_windows-any.ps1* job recipes.

The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources ***common_unix-any.sh*** or ***common_windows-any.ps1*** (represented as "***common_{unix,windows}-any.{sh,ps1}***") automataCI execution.
4. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) tech-specific customized CI job recipe's ***$PROJECT_{TECH}/.ci/clean_unix-any.sh*** or ***$PROJECT_{TECH}/.ci/clean_windows-any.ps1*.**
5. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/clean_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/clean_windows-any.ps1*.**

9.14. Stop

Stop CI job operates by initializing the repository via AutomataCI for obtaining the reset instructions for stopping the project operational environment. Depending on technologies, some requires and provides initialization instructions like Python with its *venv* sourcing commands.

9.14.1. Operating Mechanism

The default CI Job executes in series due to its lightweight role. Parallel executions are available.

9.14.2. Customization

Stop CI job offers CI job recipe customization for both baseline (*$PROJECT_PATH_SOURCE*) and tech-specific (e.g. Go for *$PROJECT_GO*) sources. AutomataCI shall source their *.ci/stop_unix-any.sh* and *.ci/stop_windows-any.ps1* job recipes.

The execution sequences are as follows:

1. ***automataCI/ci.sh.ps1*** is being executed by user.
2. ***automataCI/ci.sh.ps1*** kicks start ***automataCI/ci.sh*** or ***automataCI/ci.ps1*** (represented as "***automataCI/ci.{sh,ps1}***") depending on host machine's OS.
3. ***automataCI/ci.{sh,ps1}*** sources ***common_unix-any.sh*** or ***common_windows-any.ps1*** (represented as "***common_{unix,windows}-any.{sh,ps1}***") automataCI execution.
4. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) tech-specific customized CI job recipe's ***$PROJECT_{TECH}/.ci/stop_unix-any.sh*** or ***$PROJECT_{TECH}/.ci/stop_windows-any.ps1*.**
5. ***common_{unix,windows}-any.{sh,ps1}*** sources (and implicitly execute) baseline customized CI job recipe's ***$PROJECT_PATH_SOURCE/.ci/stop_unix-any.sh*** or ***$PROJECT_PATH_SOURCE/.ci/stop_windows-any.ps1*.**

9.14.3. Virtual Environment Initialization

AutomataCI usually by default setup all technologies' virtual environment which may contain one or more manual instructions. Hence, please read the on-screen instructions developed by the project maintainers & developers.

9.15. Purge

Purge CI job operates by purging everything done in the repository and reset it back to its initial state. It does not remove external tools setup by ENV CI job.

WARNING

This is a nuclear button. Once executed, there is NO TURNING BACK but to run everything from scratch.

Please make ABSOLUTELY SURE with your intention before executing this CI Job.

9.15.1. Operating Mechanism

This CI Job executes in series. Parallel executions is unavailable.

9.15.2. No Customization

Due to simplicity sake, **this job does not offer any CI customization**.

10. Developing with AutomataCI

Before we begin, on behalf of AutomataCI development team, we Thank you for your interest in contributing towards AutomataCI source codes and building with us for your project. This section covers all the necessary technical specifications in order to develop for AutomataCI or for your project customization.

10.1. Tech Requirements

To be seamlessly compatible with the OS natively, the entire AutomataCI is created using only POSIX compliant shell (not BASH) or "POSIX Shell" scripts and Windows' PowerShell scripts. At its root, to make the interfaces unified, the Polygot Script (https://github.com/ChewKeanHo/PolygotScript) is used where the POSIX Shell and PowerShell are unified into a single file called "***ci.sh.ps1***".

Generally speaking, you only need the following technical knowledge to operate AutomataCI (as in developing your own CI recipes and etc):

1. **POSIX Shell Scripting (not BASH)** – for all UNIX OSes including Apple MacOS; AND
2. **Windows PowerShell** – for Windows OS.

To develop with AutomataCI, you need a very deep understanding about them as AutomataCI always cross-translate them to make sure its functionalities are working on as many systems as possible.

10.1.1. POSIX Shell

Branded as “POSIX Shell” (not to be confused with BASH shell), it is an UNIX execution scripting language meant for UNIX operating systems like Linux – Debian, BSD – Debian, OSX – Mac, and etc. To be POSIX compliant, you must first understand and familiarize yourself with BASH scripting and then works yourself towards POSIX compliant shell:

1. Things like array, lowercase string changes, double square quote comparison (*[[...]]*) are unavailable; AND
2. Use primitive shell runtime like Dash (*/bin/dash*) to validate your POSIX Shell script instead of the usual */bin/sh* which is usually pointing */bin/bash* runtime.
3. you can run it on both Linux and MacOS OSes without additional modifications and special conditions and they both work as expected.

AutomataCI prioritize Linux and MacOS OSes using POSIX compliant shell. If you’re new, reading the codes from the *automataCI/services/* libraries to obtain some good production-grade ideas.

10.1.2. Powershell

Powershell is solely used for Windows OS despite Microsoft permits its installation elsewhere. AutomataCI prohibits the latter for the fact since the installation requires Microsoft telemetry feature to be installed which breaks a perfectly fine UNIX operating system. Unlike OSX in Mac ecosystem, to date, *Microsoft Windows is still incompatible with POSIX shell without Linux subsystem (which is Linux VM-like anyway)*.

Like most Microsoft’s products pipelines, PowerShell also come with its caveats and dramas where AutomataCI had solved them in its coding style section.

10.2. Release Strategy

AutomataCI uses Semantic Versioning system (https://semver.org/) to manage its release versions. To ensure sorting compatibility, a "*v-*" prefix is added to each version number when published.

Unlike other strategies, AutomataCI approach forward increment aggressively by only supporting the latest version of itself expiring all supports to the previous versions. However, that does not means the customers have to immediately upgrade it to the latest and greatest due to AutomataCI's product nature.

Only in special (and rare) cases, AutomataCI may deploy release candidate (*~rcN*) strategy to perform external ecosystem testing where:

1. *N* is candidate number; AND
2. **always uses tilde (~) and not dash (-)** to avoid conflicting with some mainline distributing ecosystems.

REMEMBER: All release candidates ARE NOT a latest release.

10.3. Upstreaming Process

In the case of developing for AutomataCI involving upstreaming the code changes to the AutomataCI original repository, the steps are described in the sub-sections.

10.3.1. Raise an Issue Ticket in Forum

Start-off by raising an issue ticket in the designated forum (usually GitHub) and discuss about it. You will be surprised how much this helps to expedite problem solving and sometimes time, don't have to do much at all.

10.3.2. Clone the AutomataCI repository and Perform Development

Please clone the original AutomataCI repository and develop there. The upstream maintainers only accepts AutomataCI upstream repository only and shall reject any patches not from those repository.

10.3.3. Set Patches / Pull Request, Code Review

Once done, please make it available for the maintainers to code review your changes. You can git format patch your commits and upload it into your issue ticket raised earlier.

10.3.4. Acceptance

There are 2 possible acceptance outcomes depending on how you approach the maintainers:

1. If you're sent in by patch, the maintainer shall apply it to the repo and signed it using his/her GPG key; OR
2. If you're sent in by pull request, the maintainer shall perform merging locally and signed it using his/her GPG key, accepting your pull request to the upstream and then force push to override the unsigned commits.

At this point, you're done for your part. Once again, thank you for developing with AutomataCI.

10.4. Coding Styles

This section covers the coding styles that is compatible for both shells types and its rationale behinds them.

10.4.1. Understanding Shelling Nightmare

The most headache problem is to make both Powershell and POSIX Shell into a single same pattern with same context like a human language i18n translation feature. It's like trying to marry water (POSIX Shell) with oil (PowerShell) working in the same way: **they don't mix**. Let's review some of the cases encountered during the development of AutomataCI.

10.4.1.1. PowerShell – Way Too Many Feedback Channels

In POSIX Shell, all commands only has to check against $? variable that contain last execution status. PowerShell also however, has too many ways and too many ambiguous manners to check, it's hard to consistently check each of them (at least 5 combinations of probable cases depending on the command). Here's a simple case study (see next page).

```
# POSIX
some_command “...”
if [ $? -ne 0 ]; then
	# handles error
	exit 1
fi
```

Versus:

```
# POWERSHELL TYPE 1 (4 possibilities for 1 single command)
try {
	$__output = Invoke-Expression “some_command `”...`””
	if (($LASTEXITCODE -ne 0) -or ($? -ne 0) -or (-not $__output)) {
		exit 1
	}
} catch {
	# definitely error. Handle it.
}

# POWERSHELL TYPE 2 (another 1 possibility is via object-oriented method) $__process = Start-Process `
	-Wait `
	-FilePath "$__program" `
	-NoNewWindow `
	-ArgumentList "$__arguments" `
	-PassThru
if ($__process.ExitCode -ne 0) {
	# handle error
	exit 1
}
```

10.4.1.2. PowerShell – Return is Not Return

The worst of all – PowerShell's return is not behaving as commonly expected. In POSIX Shell, return always set the numeric value for *$?* variable. PowerShell returns everything inside the function output so you need to redirect every single command's output or status values into $null in order to get a clean *$?* similar pattern.

```
function OUTPUT-Function-Name {
        Invoke-Expression "dir"
        return 0
}

$__output = OUTPUT-Function-Name (Get-Location)
# $__output is not 0; it's the "$(dir output) + 0".
# Also, "0" is not a return code but is an output so $LASTEXITCODE or $? may fail.
```

10.4.1.3. PowerShell – Not all CMDlets Are Available

Remember the fact that AutomataCI prohibits PowerShell installation in UNIX OS earlier? ***This is also because not all cmdlets are available if you do so***. Journeying into this path makes creating a cross-platform PowerShell script notoriously complicated and inconsistent via guessing and gambling with fate nature.

10.4.1.4. Live with It

It's cute to find out PowerShell aspiring to outgrow itself to become a (comparable?) full-fetch programming language but the lack of architecture planning quite a bizarre left alone behaving inconsistently. Regardless, through many failed attempts, AutomataCI discovered a way to do it successfully as long as the scripts developers adhere to certain rules set by AutomataCI dealing with these arcane but powerful technologies.

10.4.2. Quality Assurance Testing and Warranted Functionalities

Unlike a proper programming language, both POSIX Shell and PowerShell lacks the facilitates to perform proper library development left alone some good test infrastructures. Fortunately, both shells runtime actually validates the entire script when performing dot import.

Hence, to ensure all functionality are definitely working (and purge all their complexities and guessing fanaticism), AutomataCI defined some strict rules to make sure everything is working as it is, across different host machines with different probable combination of CPU architectures and OS.

10.4.2.1. Functionalize Everything

To make sure the executions are re-usable (which helps in testing most of the codes themselves), all executions shall be completely functionalized complying to the following pattern (see next page):

The more reusable the library, the more resilient for all the functions contained within thanks to the shell runtime's full script validations. Hence, please keep the libraries as simple and granular as possible.

```
# POSIX Shell
LIBRARY_Function_Name_In_Underscore_Titlecase() {
        __param_name_1=”$1”
        __param_name_2=”$2”
        …
        __param_name_N=”$N”


        # validate input
        …


        # execute
        …


        # report status
        return 0
}
```

```
# POWERSHELL
LIBRARY-Function-Name-In-Dashing-Titlecase {
	param(
		[string]$__param_name_1,
		[string]$__param_name_2,
		...
		[string]$__param_name_N
	)


	# validate input
	$null = ...
	...


	# execute
	$null = ...
	...


	# report status
	return 0
}
```

That way, the calling for both types of shell functions are having close similarities:

```
# POSIX Shell
LIBRARY_Function_Name_In_Underscore_Titlecase “...” “...” ...
if [ $? -ne 0 ]; then
        0return 1
fi
```

```
# POWERSHELL
$__process = LIBRARY-Function-Name-In-Dashing-Titlecase “...” “...” ...
if ($__process -ne 0) {
        return 1
}
```

The rules:

1. Write for human to read and maintain; DO NOT dare to optimize solely for machine.
2. **[NON-COMPROMISING RULE]** Both POSIX Shell and PowerShell must share maximum similarity to the point where it can be nearly 1:1 comparing to each other.
3. **[NON-COMPROMISING RULE]** All variables MUST be inserted with a quoted string manner (e.g. “$__list” in POSIX Shell and “${__list}” in PowerShell).
4. **[NON-COMPROMISING RULE]** Maintain absolute silence at all time and only handle control via return code.
5. POSIX Shell should comply to **Underscore_Titlecase** function name while PowerShell should comply to its **Dashing-Titlecase** function name; AND
6. For PowerShell, remember to redirect all unused return values to **$null** to make sure return output is actually returning the exit code only.
7. For *LIBRARY-function* differentiation, the LIBRARY is fully **UPPERCASE** for both shells.
8. **Stick to positional parameters ONLY**. POSIX Shell does not have any alternative.
9. For each execution phases (e.g. validate input, execute, and report status), minimum 2 line spacing is required for readability purposes (add more if the function is very complex).
10. For each executions block within the same execution phase, a minimum of 1 line spacing is required for readability purposes (add more if the function is very complex).

GENTLE REMINDER

refer to existing libraries when learning before asking.

10.4.2.2. Data Processing Functions

For data processing function (e.g. returning data as output), **DO NOT return the exit code** since PowerShell cannot programmatically to do so. Instead, choose a default value to return as negative expectation and have it checked instead. Here's a deployed case study:

```
# POSIX Shell
STRINGS_To_Lowercase() {
	#__content="$1"


	# execute
	printf "$1" | tr '[:upper:]' '[:lower:]'
}
```

```
# PowerShell
function STRINGS-To-Lowercase {
	param(
		[string]$__content
	)


	# execute
	return $__content.ToLower()
}
```

Call them would be:

```
# POSIX Shell
__output=”$(STRINGS_To_Lowercase “$__data”)”
if [ -z “$__output” ]; then
        return 1
fi
```

```
# POWERSHELL
$__output = STRINGS-To-Lowercase “${__data}”
if ([string]::IsNullOrEmpty($__output)) {
        return 1
}
```

If error handling is required, use a default value (usually empty value) as an indicator instead of relying on exit code. The example above uses empty string as the indicator.

10.4.2.3. Simplifying POSIX Shell's Function Parameters

If a given function has too little parameters, you're allowed to comment out the parameters naming and use the positional parameters directly in POSIX Shell. The condition to apply this exception are as follows:

1. Only use 1 parameter; OR
2. Only use maximum of 2 parameters AND within a maximum of 10 lines long with max 3-tab complexities.

Here's a case-study:

```
# POSIX Shell
STRINGS_Has_Suffix() {
	#__suffix="$1"
	#__content="$2"


	# execute
	case "$2" in
	*"$1")
		return 0
		;;
	*)
		return 1
		;;
	esac
}
```

10.4.3. Libraries Organization

Fortunately, both shell types share the same limitations for both their functions and variables naming conventions. Understand that both shells do not have scoping capabilities so any constant, public, private scoping interpretation is strictly through naming convention. *All functions shall be re-importable as the same definitions over and over again so the naming convention plays a vital roles in libraries organization* (see later sections).

10.4.3.1. Basic Layering Concept

you SHALL do the following for both functions and variables in both shell types:

1. Comply to the lead underscoring naming convention to indicates its deepness:
    1.1. **Layer 0 lowercase** – indicates an internally used entity (e.g. *$pkg*, *$data*).
    1.2. **Layer 0 UPPERCASE** – indicates an exportable “constants” or a library suite (e.g ***STRINGS***_*Fx_Name*, ***STRINGS-****Fx-Name*, ***$PROJECT_PATH_ROOT***).
    1.3. **Layer 1 lowercase** – indicates private entity (e.g. *$_target*)
    1.4. **Layer 2 lowercase** – indicate private and frequently disposable entity (e.g. *$__ret*)
    1.5. **Layer 3 lowercase** – indicate an internally used entity inside a library.
2. **Deepness level SHALL NOT go beyond layer 3**. If you do, it signifies you have a problem with re-organization (see next section).
3. For privately scoped function, prepend the word “*Subroutine-*” to avoid naming conflict and easily called by exernal (since all functions are exported regardless).

10.4.3.2. Architectural Layers

AutomataCI's libraries are organized into multi-layers architecturally. Table 61 tabulates all the layers accordingly.

Table 61: AutomataCI Libraries Architectural Layers

Layer	File	Trigger	Descriptions
0	automataCI/ci.sh.ps1	Execute	The Polygot executable script to unify user commands independent of OS and CPU architecture.
	automataCI/ci.sh, automataCI/ci.ps1	Source	The OS-specific initialization script: .ps1 for PowerShell on Windows OS; .sh for POSIX Shell on UNIX OSes.
1	automataCI/[TYPE]	Source	the job functions.
	automataCI/_[TYPE]	Source	the job functions' subroutine modules when it's main script is too big.
2	[TECH]/[CI]/[JOB]	Source	the tech-specific job recipe that facilitates customization
3	automataCI/services/*	Source	the common function libraries' services.
4	automataCI/services/io/*	Source	The primitive function libraries' services.

As noted above, understand that everything happens in AutomataCI is using source direction with the only exception to *automataCI\ci.sh.ps1* Polygot script initialization. Hence, unless you're working on *automataCI\ci.sh.ps1*, **you shall only use "*return*" as the exit command, not "*exit*"**.

AutomataCI's Layer 1 is expected to tell the journey of the CI execution pipeline with 100% pinpoint accuracy working alongside Layer 2. ***Both Layer 1 and Layer 2 are strongly encouraged to import and use Layer 3 and Layer 4 libraries for consistency and future-proof themselves*** from low-level OS side changes.

10.4.3.3. Internationalization Supported Text Printing

It's duly noted that terminal printout (e.g. *stdout* & *stderr*) with internationalization supports is **strictly Layer 1 and Layer 2 ONLY**. Other layers are forbidden to report anything but returning an exit code.

10.4.3.4. Primitive Self-Contained Layer 4 IO Libraries

ALL Layer 4 library suites MUST BE SELF-CONTAINED. This means any library suites falls in *automataCI/services/io/* directory are primitive building blocks and shall not import anything including each others.

The main reason is to avoid diamond import pattern and circular/cyclic import problem.

10.4.3.5. Re-importable Libraries

All libraries and functions SHALL BE IMPORTABLE AND RE-IMPORTABLE.

Both shells do not have a concept of library management. Hence, each time a source code imports a library, the same function get re-imported again and again, with the latest version overriding the former. Given which such circumstances, please make sure the functions and variables are re-importable which offers identical functionalities under the same symbol.

10.4.3.6. Symbol Grouping using LIBRARIES- Prefix Convention

Every libraries SHALL use the uppercase "*LIBRARIES-*" prefix convention stated earlier to minimize symbol collisions. That way, a function with similar intention and naming can be reused while having the LIBRARIES- prefix differentiating them. Here's a case study where the function name is "*Create*" being reused across multiple libraries:

```
TAR_Create ...
ZIP_Create ...
XZ_Create ...
MSI_Create ...
```

10.4.3.7. Keep Everything in AutomataCI/ Directory

To ensure future upgrades availability and ease of distributions, ***please keep every AutomataCI stuffs inside automataCI/ directory***. As demonstrated in the quick start section, AutomataCI customers shall only update *automataCI/* directory from the upstream and perform any localized patching later.

We only want to distribute that directory and that's it.

10.4.4. Looping

Looping in POSIX Shell and PowerShell can be quite complicated while maintaining similarities between them. This section covers all the basic and commonly used looping mechanism to avoid pitfalls during development.

Keep in mind that the Internet is flooded with very bad or AI regurgitated guiding content that show poor understanding about shell runtime. Hence, whenever specified here, please abide to it.

Otherwise, you can read up GNU POSIX Shell engineering specification alongside Microsoft PowerShell engineering specification. They are thick books but this is the preferred direction (instead of the wild Internet).

10.4.4.1. Looping Over A File Content

There are so many ways to loop a file content and process them line-by-line, this is the preferred way:

```
# PowerShell
foreach ($__line in (Get-Content “/path/to/file”)) {
        Write-Host “${__line}”
}
```

```
# POSIX Shell
old_IFS="$IFS"
while IFS="" read -r __line || [ -n "$__line" ]; do
        1>&2 printf "${__line}\n"
done < "/path/to/file"
IFS="$old_IFS" && unset old_IFS
```

In POSIX Shell, it is your responsibility to save the original *$IFS* (separator) for after-use restoration (see the last line).

This method is preferred mainly because they did not use any external dependencies.

10.4.4.2. Looping Over A Directory

To loop over a directory, there are many ways to do it especially when using external commands. However, AutomataCI restricts minimum to no external dependencies for simple operations as follows:

```
# PowerShell (basic looping)
foreach ($__line in (Get-ChildItem -Path "/path/to/directory" `
                        | Select-Object -ExpandProperty FullName)) {
        # filter accordingly
        if (Test-Path -PathType Container -Path “${__line}”) {
                # it’s a directory. Handle it.
        }

        if (-not (Test-Path -PathType Container -Path “${__line}”)) {
                # it’s a directory. Handle it.
        }


        # execute
        ...
}
```

```
# PowerShell (with .log file extension filter as an example)
foreach ($__line in (Get-ChildItem -Path "/path/to/directory" -Include *.log `
                        | Select-Object -ExpandProperty FullName)) {
        ...
}
```

```
# POSIX Shell (Basic Looping)
# IMPORTANT: make sure when using any globe (*), it **MUST** be outside of string.
for __line in "/path/to/directory/"*; do
        if [ ! -e "$__line" ]; then
                continue # POSIX Shell passes the string as the last item so bail it out.
        fi



        # filter accordingly
        if [ -d "$__line" ]; then
                # it's a directory. Handle it.
        fi

        if [ -f "$__line" ]; then
                # it's a directory. Handle it.
        fi



        # execute
        ...
done
```

```
# POSIX Shell (with .log file extension filter as an example)
for __line in "/path/to/directory/"*.log; do
        ...
done
```

10.4.4.3. Looping Over A List

There are various ways to loop over a list in POSIX Shell and PowerShell but the following are preferred as the list configurations are done first (comply to top-to-bottom code review reading flow) before looping and also they are not using any external dependencies especially for POSIX Shell:

```
# PowerShell
foreach ($___line in @(
        "item1|opt1|opt2|..."
        "item1|opt1|opt2|..."
)) {
        # parse input
        $___list = $___line -split "\|"
        $___item = $___list[0]
        $___opt1 = $___list[1]
        $___opt2 = $___list[2]
        ...


        # execute
        ...
}
```

```
# POSIX Shell
# NOTE:
#   1. This is using newline (\n) as a separator so each line is a new element.
#   1.1. That also means, due to limitation, DO NOT do complicated listing.
#   2. The entire thing is a double-quoted string so variables can be used as well.
#  3. Break your line’s options in the line with another separator (e.g. “|”).
unique_naming_list="\
item1|opt1|opt2|...
item2|opt1|opt2|...
"

old_IFS="$IFS"
while IFS="" read -r ___line || [ -n "$___line" ]; do
        # parse input
        ___item="${___line%%|*}"
        ___line="${___line#*|}"

        ___opt1="${___line%%|*}"
        ___line="${___line#*|}"

        ___opt2="${___line%%|*}"
        ___line="${___line#*|}"
        ...



        # execute
        ...
done <<EOF
$unique_naming_list
EOF
IFS="$old_IFS" && unset old_IFS # always restore $IFS before doing anything else
```

11. Epilogue

That’s all from us. We wish you would enjoy the project development experiences to your delights.