# {robust-externalize}

**Cache anything (Ti*k*Z, python…),
in a robust, efficient and pure way.**

Léo Colisson Version 2023/03/22-unstable

github.com/leo-colisson/robust-externalize

## Contents

**WARNING: This library is very young and has not been tested extensively. Even if we try to stay backward compatible, the only guaranteed way to be immune to changes is to copy/paste the library in your main project folder.**

## 1 Introduction

### 1.1 Why do I need to cache (a.k.a. externalize) parts of my document?

One often wants to cache (i.e. store pre-compiled parts of the document, like figures) operations that are long to do: For instance, TikZ is great, but TikZ figures often takes time to compile (it can easily take a few seconds per picture). This can become really annoying with documents containing many pictures, as the compilation can take multiple minutes: for instance my thesis needed roughly 30mn to compile as it contains many tiny figures, and LaTeX needs to compile the document multiple times before converging to the final result. But even on much smaller documents you can easily reach a few minutes of compilation, which is not only high to get a useful feedback in real time, but worse, when using online LaTeX providers (e.g. overleaf), this can be a real pain as you are unable to process your document due to timeouts.

Similarly, you might want to cache the result of some codes, for instance a text or an image generated via python and matplotlib, without manually compiling them externally.

### 1.2 Why not using Ti*k*Z's externalize library?

Ti*k*Z has an externalize library to pre-compile these images on the first run. Even if this library is quite simple to use, it has multiple issues:

- If you add a picture before existing pre-compiled pictures, the pictures that are placed after will be recompiled from scratch. This can be mitigated by manually adding a different prefix to each picture, but it is highly not practical to use.
- To compile each picture, TikZ's externalize library reads the document's preambule and needs to process (quickly) the whole document. In large documents (or in documents relying on many packages), this can result in a significant loading time, sometimes much bigger than the time to compile the document without the externalize library: for instance, if the document takes 10 seconds to be processed, and if you have 200 pictures that take 1s each to be compiled, the first compilation with the TikZ's externalize library will take roughly half an hour instead of 3mn without the library. And if you add a single picture at the beginning of the document… you need to restart everything from scratch. For these reasons, I was not even able to compile my thesis with TikZ's external library in a reasonable time.
- If two pictures share the same code, it will be compiled twice
- Little purity is enforced: if a macro changes before a pre-compiled picture that uses this macro, the figure will not be updated. This can result in different documents depending on whether the cache is cleared or not.
- As far as I know, it is made for TikZ picture mostly, and is not really made for inserting other stuff, like matplotlib images generated from python etc...
- According to some maintainers of TikZ, "the code of the externalization library is mostly unreadable gibberish[1]", and therefore most of the above issues are unlikely to be solved in a foreseable future.

## 1.3 FAQ

**What is not supported?** We don't support (yet) overlays, remember picture, and you can't use (yet) cross-references inside your images (at least not without further hacks). See other limitations and known bugs at the end of this documentation. Note that this library is quite young, so expect untested things.

**Do I need to compile using -shell-escape?** Since we need to compile the images via an external command, the simpler option is to add the argument `-shell-escape` to let the library run the compilation command automatically (this is also the case of Ti*k*Z's externalize library). However, people worried by security issues of `-shell-escape` (that allows arbitrary code execution if you don't trust the LaTeX code) might be interested by these facts:

- If images are all already cached, you don't need to enable `-shell-escape`.
- You can choose not to compile non-cached content, and display a dummy content instead until you choose to compile them.
- You can compile manually the images: all the commands that are left to be executed are listed in `robExt-compile-missing-figures.sh` and you can just run them, either with `bash robExt-compile-missing-figures.sh` or by typing them manually (most of the time it's only a matter of running `pdflatex somefile.tex`).

**Is it working on overleaf?** Yes: overleaf automatically compiles documents with `-shell-escape`, so nothing special needs to be done there (of course, if you use this library to run some code, the programming language might not be available, but I heard that python is installed on overleaf servers for instance, even if this needs to be doubled checked). If the first compilation of the document to cache images times out, you can just repeat this operation multiple times until all images are cached.

[1] `https://github.com/pgf-tikz/pgf/issues/758`

**Do you have some benchmarks?** On an early draft of a small paper containing 76 small tikz-cd based pictures (from my other zx-calculus library), we measured:

- 35 seconds for a normal compilation without externalization
- 75 seconds for the first compilation with this library
- 2.4 seconds for the next runs

So during the first compilation, we lost a x2 factor (roughly an additional time of .5 seconds per picture coming from the time to start LaTeX, it seems like on average a picture takes .5 seconds to be built in my benchmark), but then we have a speedup of x15 (2.43s instead of 34.63s) for all subsequent runs. And I expect this to be even higher with more pictures and more complex documents.

**Can I use version-control to keep the cached files in my repository?** Sure, each cached figure is stored in a few files (typically one pdf and one LaTeX file, plus the source) having the same prefix (the hash), avoiding collision between runs. Just commit these files and you are good to go.

**Can you deal with baseline position ?** Yes, the depth of the box is automatically computed and used to include the figure by default.

**How is purity enforced?** Purity is the property that if you remove the cached files and recompile your document, you should end-up with the same output. To enforce purity, we compute the hash of the final program, including the compilation command and the dependency files used for instance in `\input{include.tex}` (unless you prefer not to, for instance to keep parts of the process impure for efficiency reasons), and put the code in a file named based on this hash. Then we compile it if it has not been used before, and include the output. Changing a single character in the file, the tracked dependencies, or the compilation command will lead to a new hash, and therefore to a new generated picture.

**Can I extend it easily?** We tried to take a quite modular approach in order to allow easy extensions. Internally, to support a new cache scheme, we only expect a string containing the program (possibly produced using a template), a list of dependencies, a command to compile this program (e.g. producing a pdf and possibly a tex file with the properties (depth. . . ) of the pdf), and a command to load the result of the compilation into the final document (called after loading the previously mentioned optional tex file). Thanks to pgfkeys, it is then possible to create simple pre-made settings to automatically apply when needed.

## 2 Quickstart

### 2.1 Installation

To install the library, just copy the `robust-externalize.sty` file into the root of the project. Then, load the library using:

```
\usepackage{robust-externalize}
```

### 2.2 Usage

#### 2.2.1 For LaTeX based content

## 3 Documentation

### 3.1 Placeholders

Placeholders are the main concept allowing this library to generate the content of a source file based on a template (a template will itself be a placeholder containing other placeholders). A placeholder is a special strings like `[°COLOR°IMAGE°]` inserted for instance in a template, that will be later given a value. This value will be used to replace (recursively) the placeholder in the template. For instance, if a placeholder `[°LIKES°]` contains `I like ._FRUIT_. and ._VEGETABLE_.]`, if the placeholder `._FRUIT_.]` contains `oranges` and if the placeholder `..VEGETABLE..]` contains `salad`, then evaluating `[°LIKES°]` will output `I like oranges and salad.`

### 3.1.1 Reading a placeholder

`\getPlaceholder[`⟨*new placeholder name*⟩`]{`⟨*name placeholder or string*⟩`}`
`\getPlaceholderInResult[`⟨*new placeholder name*⟩`]{`⟨*name placeholder or string*⟩`}`

Get the value of a placeholder after replacing (recursively) all the inner placeholders. `\getPlaceholderInResult` puts the resulting string in a LaTeX 3 string `\l_robExt_result_str`, while `\getPlaceholder` directly outputs this string. You can also put inside the argument any arbitrary string, allowing you, for instance, to concatenate multiple placeholders, copy a placeholder etc. Note that you will get a string, but this string will not be evaluated by LaTeX (see `\evalPlaceholder` for that), for instance math will not be interpreted:

The placeholder evaluates to:
`Hello Alice the great, I am a template $\delta _n$.`
Combining placeholders produces:
`In ‘‘Hello Alice the great, I am a template $\delta _n$.’’, the name is Alice the great.`

```
\placeholderFromContent{__MY_PLACEHOLDER__}{Hello __NAME__, I am a template $\delta_n$.}
\placeholderFromContent{__NAME__}{Alice __NICKNAME__}
\placeholderFromContent{__NICKNAME__}{the great}
The placeholder evaluates to:\\
\texttt{\getPlaceholder{__MY_PLACEHOLDER__}}\\
Combining placeholders produces:\\
\texttt{\getPlaceholder{In ‘‘__MY_PLACEHOLDER__’’, the name is __NAME__.}}
```

You can also specify the optional argument in order to additionally define a new placeholder containing the resulting string (but you might prefer to use its alias `\setPlaceholderRec` described below):

List of placeholders:
- Placeholder called `__NEW_PLACEHOLDER__` contains:

`In ‘‘Hello Alice the great, I am a template $\delta _n$.’’, the name is Alice the great.`

- Placeholder called `__NICKNAME__` contains:

`the great`

- Placeholder called `__NAME__` contains:

`Alice __NICKNAME__`

- Placeholder called `__MY_PLACEHOLDER__` contains:

`Hello __NAME__, I am a template $\delta _n$.`

```
\placeholderFromContent{__MY_PLACEHOLDER__}{Hello __NAME__, I am a template $\delta_n$.}
\placeholderFromContent{__NAME__}{Alice __NICKNAME__}
\placeholderFromContent{__NICKNAME__}{the great}
\getPlaceholderInResult[__NEW_PLACEHOLDER__]{In ‘‘__MY_PLACEHOLDER__’’, the name is __NAME__.}
\printAllPlaceholders
```

`\evalPlaceholder{`⟨*name placeholder or string*⟩`}`

Evaluate the value of a placeholder after replacing (recursively) all the inner placeholders. You can also put inside any arbitrary string.

The placeholder evaluates to:
Hello Alice the great, I am a template $\delta_n$.
Combining placeholders produces:
In "Hello Alice the great, I am a template $\delta_n$.", the name is Alice the great.

```
\placeholderFromContent{__MY_PLACEHOLDER__}{Hello __NAME__, I am a template $\delta_n$.}
\placeholderFromContent{__NAME__}{Alice __NICKNAME__}
\placeholderFromContent{__NICKNAME__}{the great}
% The placeholder evaluates to \texttt{\getPlaceholder{__MY_PLACEHOLDER__}}.
The placeholder evaluates to:\\
\evalPlaceholder{__MY_PLACEHOLDER__}\\
Combining placeholders produces:\\
\evalPlaceholder{In ``__MY_PLACEHOLDER__'', the name is __NAME__.}
```

### 3.1.2 List and debug placeholders

It can sometimes be handy to list all placeholders, print their contents etc. We list here commands that are mostly useful for debugging purposes.

`\printAllPlaceholders`

Prints the verbatim content of all defined placeholders (without performing any replacement of inner placeholders). This is mostly for debugging purposes.

List of placeholders:
- Placeholder called `__NAME__` contains:

```
Alice
```

- Placeholder called `__LIKES__` contains:

```
Hello __NAME__ I am a really basic template $\delta _n$.
```

```
\placeholderFromContent{__LIKES__}{Hello __NAME__ I am a really basic template $\delta_n$.}
\placeholderFromContent{__NAME__}{Alice}
\printAllPlaceholders
```

`\printPlaceholderNoReplacement{`⟨*name placeholder*⟩`}`

Prints the verbatim content of a given placeholder, without evaluating it and **without replacing inner placeholders: it is used mostly for debugging purposes** and will be used in this documentation to display the content of the placeholder for educational purposes. The stared version prints it inline.

The (unexpanded) template contains

```
Hello NAME I am a really basic template $\delta _n$.
```

.
The (unexpanded) template contains `Hello NAME I am a really basic template $\delta _n$.`

```
\placeholderFromContent{__LIKES__}{Hello NAME I am a really basic template $\delta_n$.}
\placeholderFromContent{NAME}{Alice}
The (unexpanded) template contains \printPlaceholderNoReplacement{__LIKES__}.\\
The (unexpanded) template contains \printPlaceholderNoReplacement*{__LIKES__}
```

`\evalPlaceholderNoReplacement{`⟨*name placeholder*⟩`}`

Evaluates the content of a given placeholder as a LaTeX code, **without replacing the placeholders contained inside (mostly used for debugging purposes).**

The (unexpanded) template evaluates to "Hello NAME I am a really basic template $\delta_n$.".

```
\placeholderFromContent{__LIKES__}{Hello NAME I am a really basic template $\delta_n$.}
\placeholderFromContent{NAME}{Alice}
The (unexpanded) template evaluates to ``\evalPlaceholderNoReplacement{__LIKES__}''.
```

`\getPlaceholderNoReplacement{⟨name placeholder⟩}`

Like `\evalPlaceholderNoReplacement` except that it only outputs the string without evaluating the macros inside.

The (unexpanded) template contains `Hello NAME I am a really basic template $\delta _n$`.

```
\placeholderFromContent{__LIKES__}{Hello NAME I am a really basic template $\delta_n$.}
\placeholderFromContent{NAME}{Alice}
The (unexpanded) template contains \texttt{\getPlaceholderNoReplacement{__LIKES__}}
```

### 3.1.3 Setting a value to a placeholder

`\placeholderFromContent{⟨name placeholder⟩}{⟨content placeholder⟩}`
`\setPlaceholder{⟨name placeholder⟩}{⟨content placeholder⟩}`

`\placeholderFromContent` (and its alias `\setPlaceholder`) is useful to set a value to a given placeholder.

The (unexpanded) template contains

`Hello I am a basic template with math $\delta _n$ and macros \hello`

and after evaluation and setting the value of hello, you get "Hello I am a basic template with math $\delta_n$ and macros Hello my friend!".

```
\placeholderFromContent{__LIKES__}{Hello I am a basic template with math $\delta_n$ and macros \hello}
The (unexpanded) template contains \printPlaceholderNoReplacement{__LIKES__} and %
after evaluation and setting the value of hello,%
\def\hello{Hello my friend!}%
you get ``\evalPlaceholder{__LIKES__}''.
```

As you can see, **the precise content is not exactly identical to the original string**: LaTeX comments are removed, spaces are added after macros, newlines are removed etc. While this is usually not an issue when dealing with LaTeX code, it causes some troubles when dealing with non-LaTeX code. For this reason, we define **other commands** (see for instance `PlaceholderFromCode` below) that can accept verbatim content; the downside being that LaTeX forbids usage of these verbatim commands inside other macros, so you should always define them at the top level (this seems to be fundamental to how LaTeX works, as any input to a macro gets interpreted first as a LaTeX string, losing all comments for instance). Note that this is not as restrictive as it may sound, as it is always possible to define the needed placeholders before any macro, while using them inside the macro, possibly combining them with other placeholders (defined either before or inside the macro).

But before seeing how to define placeholder containing arbitrary code, let us first see how we can define a placeholder recursively, by giving it a value based on its previous value (useful for instance in order to add stuff to it):

`\setPlaceholderRec{⟨new placeholder⟩}{⟨content with placeholder⟩}`

`\setPlaceholderRec{foo}{bar}` is actually an alias for `\getPlaceholderInResult[foo]{bar}`. Note that contrary to `\setPlaceholder`, it recursively replaces all inner placeholders. This is particularly useful to add stuff to an existing (or not) placeholder:

List of placeholders:
- Placeholder called `__MY_COMMAND__` contains:

```
pdflatex myfile
```

```latex
\setPlaceholderRec{__MY_COMMAND__}{pdflatex}
\setPlaceholderRec{__MY_COMMAND__}{__MY_COMMAND__ myfile}
\printAllPlaceholders
```

Not that the if the placeholder content contains at the end the placeholder name, we will automatically remove it to avoid infinite recursion at evaluation time. This has the benefit that you can add something to a placeholder even if this placeholder does not exists yet (in which case it will be understood as the empty string):

List of placeholders:
- Placeholder called `__COMMAND_ARGS__` contains:

```
 -l -s
```

```latex
\setPlaceholderRec{__COMMAND_ARGS__}{__COMMAND_ARGS__ -l}
\setPlaceholderRec{__COMMAND_ARGS__}{__COMMAND_ARGS__ -s}
\printAllPlaceholders
```

`\evalPlaceholderInplace{`⟨*name placeholder*⟩`}`

This command will update (inplace) the content of a macro by first replacing recursively the placeholders, and finally by expanding the LaTeX macros.

List of placeholders:
- Placeholder called `__MACRO_EVALUATED__` contains:

```
Initial value
```

- Placeholder called `__MACRO_NOT_EVALUATED__` contains:

```
\mymacro
```

Compare Initial value and Final value.

```latex
\def\mymacro{Initial value}
\placeholderFromContent{__MACRO_NOT_EVALUATED__}{\mymacro}
\placeholderFromContent{__MACRO_EVALUATED__}{\mymacro}
\evalPlaceholderInplace{__MACRO_EVALUATED__}
\printAllPlaceholders
\def\mymacro{Final value}
Compare \evalPlaceholder{__MACRO_EVALUATED__} and \evalPlaceholder{__MACRO_NOT_EVALUATED__}.
```

`\begin{PlaceholderFromCode}{`⟨*name placeholder*⟩`}`
 ⟨*environment contents*⟩
`\end{PlaceholderFromCode}`

This environment is useful to set a verbatim value to a given placeholder: the advantage is that you can put inside any code, including LaTeX comments, the downside is that you cannot use it inside macros and some environments (so you typically define it before the macros and call it inside, possibly inserting other simpler placeholders inside that you can define inside the macros).

List of placeholders:
- Placeholder called `__PYTHON_CODE__` contains:

```
def my_function(b): # this is a python code
    c = {}
    d[42] = 0
    return b
```

```
\begin{PlaceholderFromCode}{__PYTHON_CODE__}
def my_function(b): # this is a python code
    c = {}
    d[42] = 0
    return b
\end{PlaceholderFromCode}
\printAllPlaceholders
```

Note that `PlaceholderFromCode` should not be used inside other macros or inside some environments (notably the ones that need to evaluate the body of the environment, e.g. using `+b` argument or `environ`) as verbatim content is parsed first by the macro, meaning that some characters might be changed or removed. For instance, any percent character would be considered as a comment, removing the rest of the line. However, this should not be be problem if you use it outside of any macro or environment, or if you load it from a file. For instance this code:

```
\begin{PlaceholderFromCode}{__PYTHON_CODE__}
def my_function(b): # this is a python code
    c = {}
    d[42] = 0
    return b % 2
\end{PlaceholderFromCode}
\printAllPlaceholders
```

would produce:

List of placeholders:
- Placeholder called `__PYTHON_CODE__` contains:

```
def my_function(b): # this is a python code
    c = {}
    d[42] = 0
    return b % 2
```

```
\printAllPlaceholders
```

Note that of course, you can define a placeholder before a macro and call it inside (explaining how we can generate this documentation).

`\placeholderPathFromFilename{`⟨*name placeholder*⟩`}{`⟨*filename*⟩`}`

`\placeholderPathFromFilename{__MYLIB__}{mylib.py}` will copy `mylib.py` in the cache (setting its hash depending on its content), and set the content of the placeholder `__MYLIB__` to the **path** of the library in the cache. Note that the path is relative to the cache folder (it is easier to use for instance if you want to call this library from a code already in the cache).

List of placeholders:
- Placeholder called `__MYLIB__` contains:

```
robExt-F6666F86DB0ACE43E817A7EB3729FA56mylib.py
```

You can also get the path relative to the root folder:
robustExternalize/robExt-F6666F86DB0ACE43E817A7EB3729FA56mylib.py

```
  \placeholderPathFromFilename{__MYLIB__}{mylib.py}
  \printAllPlaceholders
  You can also get the path relative to the root folder:\\
  \robExtAddPrefixPath{\getPlaceholderNoReplacement{__MYLIB__}}
```

`\placeholderFromFilename{`⟨*name placeholder*⟩`}{`⟨*filename*⟩`}`

`\placeholderFromFilename{__MYLIB__}{mylib.py}` will set the content of the placeholder `__MYLIB__` to the content of `mylib.py`.

List of placeholders:
- Placeholder called `__MYLIB__` contains:

```
def mylib():
    # Some comments
    a = b % 2
    return "Hello world"
```

```
\placeholderFromFilename{__MYLIB__}{mylib.py}
\printAllPlaceholders
```

`\placeholderPathFromContent{`⟨*name placeholder*⟩`}[`⟨*suffix*⟩`]{`⟨*content*⟩`}`

`\placeholderPathFromContent{__MYLIB__}{some content}` will copy `some content` in a file in the cache (setting its hash depending on its content, the filename will end with `suffix` that defaults to `.tex`), and set the content of the placeholder `__MYLIB__` to the **path** of the file in the cache. Note that the path is relative to the cache folder (it is easier to use for instance if you want to call this library from a code already in the cache).

List of placeholders:
- Placeholder called `__MYLIB__` contains:

`robExt-AC364A656060BFF5643DD21EAF3B64E6.py`

You can also get the path relative to the root folder:
robustExternalize/robExt-AC364A656060BFF5643DD21EAF3B64E6.py
As a sanity check, this file contains

```
some contents b
```

```
\placeholderPathFromContent{__MYLIB__}[.py]{some contents b}
\printAllPlaceholders
You can also get the path relative to the root folder:\\
\robExtAddPrefixPath{\getPlaceholderNoReplacement{__MYLIB__}}\\
As a sanity check, this file contains
\verbatiminput{\robExtAddPrefixPath{\getPlaceholderNoReplacement{__MYLIB__}}}
```

`\begin{PlaceholderPathFromCode}[`⟨*suffix*⟩`]{`⟨*name placeholder*⟩`}`
⟨*environment contents*⟩
`\end{PlaceholderPathFromCode}`

This environment is similar to `\placeholderPathFromContent` except that it accepts verbatim code (therefore LaTeX comments, newlines etc. will not be removed). However, due to LaTeX limitations, this environment cannot be used inside macros or some environments, or this property will not be preserved. For instance, if you create your placeholder using:

```
\begin{PlaceholderPathFromCode}[.py]{__MYLIB__}
def my_function(b): # this is a python code
    c = {}
    d[42] = 0
    return b % 2
\end{PlaceholderPathFromCode}
```

You can then use it like:

List of placeholders:
- Placeholder called `__MYLIB__` contains:

`robExt-CDAE704490400F29B9C0C8DAE2CC48B7.py`

You can also get the path relative to the root folder:
robustExternalize/robExt-CDAE704490400F29B9C0C8DAE2CC48B7.py
As a sanity check, this file contains

```
def my_function(b): # this is a python code
    c = {}
    d[42] = 0
    return b % 2
```

```
\printAllPlaceholders
You can also get the path relative to the root folder:\\
\robExtAddPrefixPath{\getPlaceholderNoReplacement{__MYLIB__}}\\
As a sanity check, this file contains
\verbatiminput{\robExtAddPrefixPath{\getPlaceholderNoReplacement{__MYLIB__}}}
```

`\placeholderFromFilename{`⟨*name placeholder*⟩`}{`⟨*filename*⟩`}`

`\placeholderFromFilename{__MYLIB__}{mylib.py}` will set the content of the placeholder `__MYLIB__` to the content of `mylib.py`.

List of placeholders:
- Placeholder called `__MYLIB__` contains:

```
def mylib():
    # Some comments
    a = b % 2
    return "Hello world"
```

```
  \placeholderFromFilename{__MYLIB__}{mylib.py}
  \printAllPlaceholders
```