The runcode package – troubleshooting

Haim Bar and HaiYing Wang

June 13, 2026

When using the runcode package, you have to be aware of some usage rules, so this document attempts to anticipate all the possible user or system errors, and show how to interpret the output and fix the problems.

Before you start using runcode

Before using the package make sure that you have Python (version 3.x) installed and in your path. Also, you must install the required LaTeX packages: morewrites, tcolorbox, xcolor, inputenc, textgreek, filecontents, xifthen, xparse, xstring, and fvextra. The minted package is optional but recommended.

If you try to compile this file from its tex source, you will get errors (because the purpose of this document is to demonstrate errors and how to fix them.) It is assumed that you are using a cloned runcode repository, so files used in this document are ones in the directory structure as it is stored on github.

The files generated by runcode live in the project's directory, so we assume the user has the necessary permissions to create, modify, delete files and subfolder.

For the server mode the folder must contain a configuration file for each statistical language used (R, julia, matlab, Python). If such files don't exist, runcode will create them with default values. Remember to check if the defaults work for you. For example, you may need to change the port number, or increase the timeout parameter.

Some possible errors and solutions

Problem:
runcode functions are not executed.

Possible reason and solution:
Check if you enabled the `shell-escape` option when the document is compiled. If not, you will see warnings in the project's log file. For example:

```
Package ifplatform Warning:
    shell escape is disabled, so I can only detect \ifwindows.
! Package minted Error: You must invoke LaTeX with the -shell-escape flag.
```

Possible reason and solution:
Check if the command-line tools you invoke from runcode are installed, and in your path (e.g., R, Julia, Matlab, Python).

Notes: The runcode package can call any command-line function when it is used in 'batch-mode'. That is, when the command-line tool is called separately each time a computation is performed from within the tex document (upon compilation). When using R, Julia, Python, or Matlab, the user can maintain a continuous session to the corresponding command-line tool. This saves initialization time, and allows to keep a session's history, for performing steps sequentially and efficiently. This is the recommended way to use runcode with R, Julia, Python, and Matlab. At the present time, no other languages are supported for 'server-mode' operation.

Problem:
Code highlighting is not working properly.

Possible reason and solution:
When code is included in the manuscript, it is done via the `\showCode` command. By default, the code-highlighting it done via the minted package. If code highlighting doesn't work, check if the minted package is installed properly. Python (3.x) also has to be installed, and also the Pygments package (which has to be installed via pip3). If you have trouble with the installation of minted, use the `nominted` option when you include the runcode package. This will cause runcode to use the fvextra for code display, instead.

Problem:
Embedded code is not shown.

Possible reason and solution:
Check if you specified the source file correctly. `\showCode` prints the source code, using minted for a pretty layout. It takes 4 arguments. Arg #1 is the programming language, Arg #2 is the source file name, Args #3 and #4 are the first and last line to show (optional). If the source file name does not exist, you will get a red and bold error message. For example:

```
\showCode{R}{Sim23.R}
```

showCode: File *Sim23.R* does not exist!
In contrast, when the file exists, as in this example

```
\showCode{R}{paper/supplement/Code/code1.R}
```

the file will be shown correctly:

```r
set.seed(0) ## fix the random number
x = rnorm(100)
y = 1+x+rnorm(100)
fit = lm(y~x)
print(summary(fit))
```

Possible reason and solution:
If the programming language is misspecified or not recognized by minted or fvextra, the code highlighting may not be shown correctly.

```
\showCode{matlab}{paper/supplement/Code/code1.R}
```

```r
set.seed(0) ## fix the random number
x = rnorm(100)
y = 1+x+rnorm(100)
fit = lm(y~x)
print(summary(fit))
```

Possible reason and solution:
If the line number in Arg #3 exceeds the actual number of lines in the code, the code box will be empty, and the LaTeX compiler will show an error message in its log file ('Empty verbatim environment'). For example:

```
\showCode{R}{paper/supplement/Code/code1.R}[6][8]
```

Some LaTeX compilers will stop the compilation when they encounter 'Empty verbatim environment' but will allow you to manually continue the compilation (and the generated pdf will contain an empty box with no code in it.)
If the number in Arg. #4 is greater than the number of lines in the file, minted will show the code up to the last line (so this misspecification is harmless).

```
\showCode{R}{paper/supplement/Code/code1.R}[4][8]
```

```r
fit = lm(y~x)
print(summary(fit))
```

Notes: Remember that the compiler is case-sensitive, so test.R is not the same as Test.R.

Problem:
Errors when running code.

Possible reason and solution:
In batch-mode with `\runExtCode`, if the source file name does not exist, you will get a red and bold error message:
`\runExtCode{julia}{test2.jl}{test2}`
runExtCode: File test2.jl does not exist!
Again, check for spelling errors in the file name, which is the most common reason for such problems.

Similarly, when using the *server-mode* of runcode, we can use the shortcuts to R, Julia, Python, or Matlab instead of `\runExtCode`. The usage is similar, but the language name is inferred from the command. For example, we can have:
`\runR{paper/supplement/Code/code1.R}{testWithrunR}`
If the source code file doesn't exist, we get an error message as with `\runExtCode`
`\runJulia{test2.jl}{test2}`
runExtCode: File test2.jl does not exist!

Possible reason and solution:
Check that the correct executable is used, since `\runExtCode` requires the specific program and command line arguments. For example, using R as the executable will not work in the following example:
`\runExtCode{R}{paper/supplement/Code/code1.R}{testWithR}`
It will create the file generated/testWithR.tex, and in it you will see
`ARGUMENT 'paper/supplement/Code/code1.R' __ignored__`
This is a fatal error – the compiler will get stuck because it will wait for the code to finish, and the compilation process will have to be terminated by the user. The correct way to use `\runExtCode` with R in batch-mode is
`\runExtCode{Rscript --save --restore}{paper/supplement/Code/code1.R}{test}`
See the package's documentation for more working examples.

Possible reason and solution:
Check for syntax errors int the code. In the *server-mode* only, `\inln`, `\inlnR`, `\inlnJulia`, `\inlnPython`, and `\inlnMatlab` commands can be used to execute short source code and embed the resulting output within the text. It takes 3 arguments. Arg #1 is the executable program; Arg #2 is the source code Arg #3 is the type output (if skipped or with empty value the default type is inline; vbox = verbatim in a box).
If the code contains a syntax error, the error produced by the statistical software will be embedded in the text. For example, if we use Factorial, instead of the real function (factorial) we get an error message:
`The factorial of 6 is \inlnR{```cat(Factorial(6))```}.`
would produce the following:
The factorial of 6 is Error in Factorial(6) : could not find function "Factorial".

Such errors are not due to LaTeX syntax or compilation, so runcode doesn't highlight them. Automatically identifying and highlighting such errors would require case-by-case analysis of the output of specific command-line tools. For now, it's up to the user to check the code before it is embedded in the tex document, and to check the output after the compilation.

Notes: Recall that `\runExtCode` which is used to run an external code takes 4 arguments: Arg #1 is the executable program, Arg #2 is the source file name, Arg #3 is the output file name (optional - if not given, the counter codeOutput is used). Arg #4 controls when to run the code (optional - if not given, it listens to `\runcode`; run = force the code to run; cache or anything else = use cache). The first argument can be *any* command-line executable, including user-defined program names (compiled code, aliases, etc.). Because of that, we do not perform validity check before trying to execute it. For example, the following will not produce any error message:
`\runExtCode{C}{test.R}{}`
but it will be possible to see an empty file in the generated folder. This will be obvious to the user once the `\includeOutput` command is executed in order to embed results in the compiled pdf document.

Problem:
Output is not produced, or unexpected/incorrect results are shown.

Possible reason and solution:
This could happen due to incorrect usage of the `\includeOutput` function, which is used to embed the output from executed code. It takes 2 arguments: Arg #1 is the output file name (optional - if not given, the counter codeOutput is used). Arg #2 is the optional type output with default "vbox" (vbox = verbatim in a box, tex = pure latex, or inline). For example, if we run the code
`\runR{paper/supplement/Code/code1.R}{testWithrunR}`
Then to see the output, the correct usage is
`\includeOutput{testWithrunR}[vbox]`

```
Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max
-1.5900 -0.8153 -0.1531  0.6379  2.8379

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.95130    0.09629    9.88   <2e-16 ***
x            1.13879    0.10960   10.39   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9626 on 98 degrees of freedom
Multiple R-squared:  0.5242,        Adjusted R-squared:  0.5193
F-statistic:   108 on 1 and 98 DF,  p-value: < 2.2e-16
```

Although the physical file is named generated/testWithrunR.tex, we drop the generated/ prefix and the .tex suffix. If we do include them, the file will not be found and we'll get an error message:
`\includeOutput{testWithrunR.tex}[vbox]`

Output file *generated/testWithrunR.tex.tex* **not found**. Check the file name (it may be that the file name was given with the suffix .tex. If so, remove it). If the file name is correct the problem may be because code execution is disabled and no cache is available. If so, force the code to run again (using the [run] option).

Notes: If anything other than vbox, tex, or inline is provided in the square brackets when using `\includeOutput`, the output will be embedded in the document as plain text. While no error or warning is raised, this may cause problems in the compilation of the tex file and therefore should be avoided. (For example, if the output contains underscores then the compiler will report an error because it would appear that math symbols are used in text mode).

Possible reason and solution:
When using one of the `inln` functions, the code does not produce any output. In some cases it is perfectly fine if embedded code does not produce output, but when using `\inln` this is not the case, so runcode checks if the command used within `\inln` produced a zero-byte output file. If it did, runcode will show an appropriate message, like in the following example. Note that the reason no output is produced is that file.csv does not exist.
`The number of columns is \inlnR{```dat <- read.csv("file.csv"); cat(ncol(dat))```}.`
This will result in:
The number of columns is **ZERO BYTES IN OUTPUT**.

Possible reason and solution:
When using one of the `inln` functions, more than one line of output is produced by the code. The `\inln` commands are designed to put a single string in a line, so the code which produces the output should not include a new line. For example, the following is incorrect:
`The factorial of 6 is \inlnR{```cat(factorial(6),"\\n")```}.`
The factorial of 6 is **ZERO BYTES IN OUTPUT**. The correct way is:

`The factorial of 6 is \inlnR{```cat(factorial(6))```}.`
which produces the following: The factorial of 6 is 720.
Similarly, if we want to print more than a one-line string, we should use the vbox option (or use `\includeOutput`.) For example,
`A matrix \inlnR{```matrix(1:9,ncol=3,nrow=3)```}.`
Will appear like this: A matrix [,1] [,2] [,3] [1,] 1 4 7 [2,] 2 5 8 [3,] 3 6 9.
`A matrix \inlnR{```matrix(1:9,ncol=3,nrow=3)```}[vbox].`
Will look better: A matrix 2026-06-13 00:04:54,952 : CRITICAL : read$_config$: $Configfile.//R.configfound.$
Possible reason and solution:
This type of problem can also be due to invalid output in the cache files. If for some reason, the cache files contain special characters or an underscore in text-mode, then the output will be corrupted.

As of v2.5, runcode automatically invalidates the cache whenever the source file changes (using an MD5 checksum), so simply editing the source file and recompiling is usually enough. If you need to force all code to re-execute regardless of whether the source has changed, use the `run` flag on the individual command or load the package with the `run` option:

```
\usepackage[run]{runcode}
```

If the cache files are corrupt (e.g., contain garbled output), remove them manually and restart the server. For example, suppose that the directory of the project is called proj, and we're using Python in server-mode. From the command line, run the following:

```
cd proj
rm -f proj.aux generated/*.txt nohup.out proj.synctex.gz
python3 -c 'from talk2stat.talk2stat import client; client("./","python","QUIT")'
rm -f serverPIDpython.txt pythondebug.txt talk2stat.log
```

Then, recompile the project.

Possible reason and solution:
Invalid, missing, or unexpected output can be due to having another runcode running on the system, but in another directory. If this happens, you will see a message in the log file saying that the server is already running. When multiple projects attempt to connect with the statistical software through the same port, only the first invocation will succeed but the subsequent ones will be connected to the first one. This will create unexpected results for all projects using the same port. Suppose your new project is B, and the runcode server of project A is still running and using the same port. In this case you may, for example,

- Accidentally overwrite variables in project A, or use the wrong ones in project B if they were also defined in A;
- Try to run code which is stored in project B, but you get an error message in the pdf file such as '`cannot open file 'mycode.R': No such file or directory`'. The reason for this error is that the other instance of the runcode server uses project A's base directory, and the file mycode.R is not in the right path.

To fix it, you can do one of the following:

1. configure each project to use another port to communicate with talk2stat (which will prevent any conflict between the projects), or,
2. you can stop all instances of talk2stat running on the system before compiling your current project.

To do the latter, find all the talk2stat processes, and find the corresponding directories. For example, suppose that multiple projects use talk2stat to communicate with Julia. From the command-line, do the following:

```
cd proj
ps -ef | grep talk2stat   # get all the PIDs. E.g., 36797
# on Mac:
lsof -a -d cwd -p  36797
# on Linux:
pwdx 36797
```

```
# lsof or pwdx will give you the directory name of the other projects which run talk2stat
# cd to each directory, and run:
python3 -c 'from talk2stat.talk2stat import client; client("./","julia","QUIT")'
rm -f serverPIDjulia.txt juliadebug.txt talk2stat.log
```

Possible reason and solution:
If the correct output is in the files in generated directory, but it's not showing in the pdf file, it can be due to the LaTeX compiler's naming of temporary files. Usually, the auto-generated file names start with the main tex file name (e.g. if the main tex file is called troubleshoot.tex then runcode will create files such as `troubleshoot_inln5.tex`). However, working with Overleaf we noticed that their compiler uses another convention for temporary file names. To prevent such problems, you can add the following in the preamble of the main document:

```
\edef\TeXjobname{\jobname} % (this line is not really essential.)
\edef\jobname{\detokenize{troubleshoot}}
```

Problem:
Working in server-mode, the server is stopped after each compilation.

Possible reason and solution:
If you are using the server-mode, be aware that some editors terminate all child processes at the end of LaTeXcompilation. For example, Emacs with Auctex behaves this way. For this case, use the nohup option, and the server will not be terminated by the parent process.
If you want the server to be stopped after each compilation (regardless of the editor you are using), use the stopserver option. While you are compiling the tex document often, you may want to keep the server side running in order to save time during initialization, and to maintain the variables and results already in memory. However, it is a good idea to use the stopserver option when you are done, just to prevent any conflicts (see above).
It's recommended to check that port numbers are unique, and when a project is not expected to be compiled for a while to enable the stopserver option.

Problem:
Using `nohup` mode, the first code block produces no output or an error, even though the source file is correct.

Possible reason and solution:
With the `nohup` option the server is started in the background, and LaTeX begins executing `\run*` commands immediately. If the server has not yet opened its socket by the time the first command is sent, that command is silently dropped and its output file is never created, causing a missing-output error later in the document.

The fix is to wait until the server is ready before the first LaTeX pass. The `wait_for_server.py` script (distributed with the package) polls the server once per second until it accepts connections:

```
python3 wait_for_server.py [LANG [DIR [TIMEOUT]]]
```

For example, to wait up to 120 seconds for an R server in the current directory:

```
python3 wait_for_server.py R ./ 120
```

Call this script between starting the server and invoking the LaTeX compiler. The `runcode-Makefile.sample` file (also distributed with the package) encodes this pattern in a ready-to-use `Makefile`.

Problem:
The process seems to hang during code execution when the LaTeX document is compiled.

Possible reason and solution:
We mentioned that code being sent to the software must be syntactically valid. In addition, keep in mind that the server waits for complete commands. So, for example, if we try
`The factorial of 6 is \inlnR{```cat(factorial(6)```}.`
it will cause the server to hang because it awaits the closing parenthesis. Eventually, it will time out but the code cannot be executed, so the desired output will not be produced, and an error message will replace the output, as in the following example. The timeout interval is set in the config files

for R, Julia, Python, and Matlab. `The factorial of 6 is \inlnR{```cat(factorial(6)```}.`
The factorial of 6 is [,1] [,2] [,3] [1,] 1 4 7 [2,] 2 5 8 [3,] 3 6 9.
In this example the cat statement is missing a closing parenthesis.