Writing a Simple Vim Plugin for REPL Interaction

Ronald G. Thomas

2025-05-20

Figure 1: A terminal split between a Vim editing buffer and a running R session with code being sent between them

A lightweight Vim plugin for sending R code to a terminal REPL.

Introduction

I did not really know how Vim's terminal API worked until I sat down and wrote a small plugin that sends code from an editing buffer to a running R session. The established tools for this task (Nvim-R and vim-slime) are feature-rich but complex. I wanted something minimal: select code in Vim, press a key, and see it execute in an R terminal pane.

The result is `rgt-R.vim`, a single-file plugin under 150 lines that handles line submission, visual selection, chunk navigation, and a handful of convenience mappings for common R inspection commands. It uses Vim's built-in `term_sendkeys()` and `term_list()` functions, requiring no external dependencies beyond Vim 8+ with terminal support.

This post presents the complete plugin code, explains each function, documents the key mappings, and discusses the experiments that shaped the design.

> **Note**
>
> This post documents the plugin's origin: a 150-line proof of concept written to understand Vim's terminal API. It has since grown into zzvim-R, a publicly released, actively maintained plugin with context-aware code submission, multi-terminal sessions, inline plot display, and a themis-based test suite. See Introducing zzvim-R for the current plugin. The code below reflects what existed in May 2025, not the present state of the project.

More formally, this post documents the Vim-plugins layer (Layer 6) of the Workflow Construct described in the Workflow Construct overview from the authorship side: how a plugin is written, not which plugins to install. This is the construct's editor-extensibility companion to Setting Up Neovim as a Data Science IDE (which documents specific R-plus-LaTeX plugins and UltiSnips snippet authorship). Together the posts give the reader the full picture of the Vim-plugins layer: which plugins to use, how to write one, and how to extend one with custom snippets.

Motivations

- Nvim-R is powerful but introduces substantial complexity for users who only need code submission; I wanted a lightweight alternative.
- vim-slime uses tmux or screen as intermediaries, adding configuration overhead that I wanted to avoid for a simple R workflow.
- Understanding Vim's terminal API (`term_sendkeys`, `term_list`) seemed valuable for building custom development workflows beyond R.
- I teach R programming and wanted a plugin simple enough that students could read and understand the source code in a single sitting.
- I was curious whether a minimal plugin could cover 90% of my daily R interaction needs without the remaining 10% of features that add complexity.

Objectives

1. Build a Vim plugin that sends individual lines, visual selections, and R Markdown chunks to a running R terminal using Vim's built-in terminal API.
2. Document each VimScript function with its purpose, mechanism, and interaction with the terminal buffer.
3. Explain the autocommand group that restricts mappings to R, Rmd, and Qmd filetypes.
4. Present the experiments that explored edge cases in visual selection submission and Quarto rendering.

This learning process is documented here. Errors spotted or better approaches are always welcome.

Understanding Vim's terminal API opens the door to custom development workflows.

Prerequisites and Setup

To use this plugin, one will need:

- Vim 8.0 or later with `+terminal` feature (check with `:echo has('terminal')`)
- R installed and accessible from the command line
- The plugin file placed in the Vim plugin directory (e.g., `~/.vim/plugin/rgt-R.vim`)

Background assumed: Basic Vim fluency (normal mode, visual mode, command mode) and familiarity with R. No prior VimScript experience is required; this post explains each function.

To verify terminal support:

```
:echo has('terminal')
```

If this returns `1`, your Vim build supports the terminal API. If it returns `0`, a Vim build with terminal support must be installed (e.g., `brew install vim` on macOS).

What is a Vim REPL Plugin?

A REPL (Read-Eval-Print Loop) plugin connects a text editor to a running interpreter. Code is written in the editor, a key combination sends it to the interpreter for execution, the interpreter prints its output, and editing continues.

Vim's terminal API provides the plumbing for this connection. The function `term_list()` returns a list of open terminal buffers, and `term_sendkeys()` sends keystrokes to a specific terminal. By combining these with Vim's visual selection and search capabilities, a small plugin can provide a complete code-submission workflow.

Think of it as a pipe between the editor and the R session. The plugin handles the mechanics of extracting selected text and pushing it through the pipe; R handles the evaluation and output.

Getting Started

Launching an R Terminal

The plugin includes a mapping to open an R terminal in a vertical split. Press `<localleader>r` in normal mode:

```
nnoremap <silent> <localleader>r
  \ :vert term R --no-save<CR><c-w>:wincmd p<CR>
```

This opens R in a vertical terminal split and returns focus to the editing buffer. The `--no-save` flag prevents R from prompting to save the workspace on exit.

The Complete Plugin

Core Submission Functions

SubmitLine: Send the Current Line

```
function! SubmitLine()
  :let @c = getline(".") . "\n"
  :call term_sendkeys(term_list()[0], @c)
endfunction
```

`SubmitLine` copies the current line into register `c`, appends a newline character, and sends it to the first terminal in `term_list()`. The newline triggers R to evaluate the expression.

Mapping: Press `<CR>` (Enter) in normal mode on any R, Rmd, or Qmd file to submit the current line.

GetVisualSelection: Extract Selected Text

```
function! GetVisualSelection(mode)
  let [line_start, column_start] =
    \ getpos("'<")[1:2]
  let [line_end, column_end] =
    \ getpos("'>")[1:2]
  let lines = getline(line_start, line_end)
  if a:mode ==# 'v'
    let lines[-1] = lines[-1][
      \ : column_end -
      \ (&selection == 'inclusive' ? 1 : 2)]
    let lines[0] = lines[0][column_start - 1:]
```

```
  elseif a:mode ==# 'V'
  else
    return ''
  endif
  return join(lines, "\n")
endfunction
```

`GetVisualSelection` handles both character-wise (`v`) and line-wise (`V`) visual selections. For character-wise selection, it trims to the exact column range. For line-wise selection, it returns the full lines without modification. The function returns the selected text as a single string with newlines preserved.

Sel and Submit: Two-Stage Submission

```
function! Sel()
  :let @c = GetVisualSelection(
    \ visualmode()) . "\n"
  :call writefile(
    \ getreg('c', 1, 1), "source_visual")
endfunction

function! Submit()
  :let y = "source('source_visual',echo=T)"
    \ . "\n"
  :call term_sendkeys(term_list()[0], y)
endfunction
```

Visual selection submission uses a two-stage process: `Sel` writes the selected text to a temporary file (`source_visual`), and `Submit` tells R to source that file with echo enabled. This approach avoids the complexity of sending multi-line text directly through `term_sendkeys`, which can cause timing issues with long selections.

Mapping: Select code visually and press `<CR>` to submit the selection.

Brk: Interrupt R

```
function! Brk()
  :call term_sendkeys(
    \ term_list()[0], "\<c-c>")
endfunction
```

`Brk` sends Ctrl-C to the R terminal, interrupting any running computation. Mapped to `<local-leader>c`.

Chunk Navigation Functions

SelectChunk: Select an R Markdown Chunk

```
function! SelectChunk()
  :execute
    \ "normal! ?```{\<cr>jV/```\<cr>k"
endfunction
```

`SelectChunk` searches backward for a code fence opening (```` ```{ ````), moves down one line, enters visual line mode, searches forward for the closing fence, and moves up one line. The result is a visual selection of the chunk contents, excluding the fences.

MoveNextChunk and MovePrevChunk

```
function! MoveNextChunk()
  :execute "normal! /```{\<CR>j"
  :noh
endfunction

function! MovePrevChunk()
  :execute "normal! 2?```{\<CR>j"
  :noh
endfunction
```

These navigation functions move the cursor to the first line of the next or previous code chunk. The `:noh` command clears the search highlighting. `MovePrevChunk` searches backward twice because the first match is the current chunk's opening fence.

Mappings: `<localleader>j` for next chunk, `<localleader>k` for previous chunk.

Convenience Functions

Raction: Quick R Inspection

```
function! Raction(action)
  :let @c = expand("<cword>")
  :let @d = a:action . "(" . @c . ")\n"
  :call term_sendkeys(term_list()[0], @d)
endfunction
```

`Raction` takes an R function name as its argument, wraps the word under the cursor as its argument, and sends the call to R. This enables quick inspection mappings:

Mapping	R Command	Purpose
`<localleader>d`	`dim(x)`	Dimensions
`<localleader>h`	`head(x)`	First rows
`<localleader>s`	`str(x)`	Structure
`<localleader>p`	`print(x)`	Print
`<localleader>n`	`names(x)`	Column names
`<localleader>f`	`length(x)`	Length

SubmitEmbed and Rd: Inline Output

```
function! SubmitEmbed()
  :let y = "sink('temp.txt'); " .
    \ "source('source_visual',echo=T); " .
    \ "sink()" . "\n"
  :call term_sendkeys(term_list()[0], y)
endfunction

function! Rd()
  !sed 's/^/\# /g' temp.txt > temp_commented.txt
  :r !cat temp_commented.txt
endfunction
```

`SubmitEmbed` sources a visual selection while redirecting output to a file. `Rd` reads that output back into the editing buffer, prepending each line with `#` so the output appears as R comments. This is useful for embedding results directly in scripts.

Mapping: `<localleader>z` in visual mode.

Figure 2: Two vintage rotary telephones connected by a coiled cord, one receiver lifted off the hook: a direct, live line between the editor and a running session

The plugin in action: code flows from the editor to the R terminal with a single keypress.

The Autocommand Group

The complete autocommand group restricts all mappings to R, Rmd, and Qmd filetypes:

```
augroup r_rmd_qmd
  autocmd!
  autocmd FileType r,rmd,qmd
    \ nnoremap <silent> <CR>
    \ :call SubmitLine()<CR><CR>
  autocmd FileType r,rmd,qmd
    \ vnoremap <silent> <CR>
    \ :call Sel() \|
    \ :call Submit()<CR><CR>
  autocmd FileType r,rmd,qmd
    \ nnoremap <silent> <localleader>c
    \ :call Brk()<CR><CR>
  autocmd FileType r,rmd,qmd
    \ nnoremap <silent> <localleader>l
    \ :call SelectChunk()<CR> \|
    \ :call Sel() \|
    \ :call Submit()<CR><CR>
  autocmd FileType r,rmd,qmd
```

```
  \ nnoremap <silent> <localleader>;
  \ :call SelectChunk()<CR> \|
  \ :call Sel() \|
  \ :call Submit()<CR> \|
  \ /```{<CR>j
autocmd FileType r,rmd,qmd
  \ nnoremap <localleader>k
  \ :call MovePrevChunk()<CR>
autocmd FileType r,rmd,qmd
  \ nnoremap <localleader>j
  \ :call MoveNextChunk()<CR>
autocmd FileType r,rmd,qmd
  \ nnoremap <silent> <localleader>r
  \ :vert term R --no-save<CR>
  \ <c-w>:wincmd p<CR>
autocmd FileType r,rmd,qmd
  \ nnoremap ZT :!R --quiet -e
  \ 'render("<C-r>%",
  \ output_format="pdf_document")'<CR>
autocmd FileType r,rmd,qmd
  \ nnoremap ZY :!R --quiet -e
  \ 'quarto_render("<C-r>%",
  \ output_format="pdf")'<CR>
autocmd FileType r,rmd,qmd
  \ tnoremap ZD
  \ quarto::quarto_render(
  \ output_format = "pdf")<CR>
autocmd FileType r,rmd,qmd
  \ tnoremap ZO source("<C-W>"%")<CR>
autocmd FileType r,rmd,qmd
  \ tnoremap ZR render("<C-W>"%")<CR>
autocmd FileType r,rmd,qmd
  \ tnoremap ZS style_dir()<CR>
autocmd FileType r,rmd,qmd
  \ tnoremap ZQ q('no')<C-\><C-n>:q!<CR>
autocmd FileType r,rmd,qmd
  \ tnoremap ZZ q('no')<C-\><C-n>:q!<CR>
autocmd FileType r,rmd,qmd
  \ tnoremap lf ls()<CR>
autocmd FileType r,rmd,qmd
  \ nnoremap <localleader>d
  \ :call Raction("dim")<CR>
autocmd FileType r,rmd,qmd
  \ nnoremap <localleader>h
  \ :call Raction("head")<CR>
autocmd FileType r,rmd,qmd
  \ nnoremap <localleader>s
```

```
    \ :call Raction("str")<CR>
  autocmd FileType r,rmd,qmd
    \ nnoremap <localleader>p
    \ :call Raction("print")<CR>
  autocmd FileType r,rmd,qmd
    \ nnoremap <localleader>n
    \ :call Raction("names")<CR>
  autocmd FileType r,rmd,qmd
    \ nnoremap <localleader>f
    \ :call Raction("length")<CR>
  autocmd FileType r,rmd,qmd
    \ inoremap <c-l>
    \ <esc>A \|><CR><C-o>0<space><space>
  autocmd FileType r,rmd,qmd
    \ vnoremap <silent> <localleader>z
    \ :call Sel() \|
    \ :call SubmitEmbed() \|
    \ :call Rd()<CR><CR>
augroup END
```

The `autocmd!` at the top clears any previous definitions, preventing duplicate mappings when the plugin is reloaded. Each mapping is scoped to R, Rmd, and Qmd filetypes only.

Experiments

Experiment 1: Adding Quarto Rendering

The `ZY` mapping was added to support rendering Quarto documents to PDF directly from Vim:

```
nnoremap ZY :!R --quiet -e
  \ 'quarto_render("<C-r>%",
  \ output_format="pdf")'<CR>
```

The development process:

1. Start by constructing the mapping in `.vimrc` (easier to iterate than in the plugin file).
2. Use `ZT` (the existing Rmd render mapping) as a template.
3. Replace `render()` with `quarto_render()` and adjust the output format argument.
4. Test with any `index.qmd` file.
5. Once working, move the mapping to the plugin file with an appropriate autocommand.

Experiment 2: Visual Selection Duplication

An investigation into why `Sel()` was sending multiple copies of the source command to R when submitting visual selections. The number of duplicate submissions equalled the number of lines in the selection.

The root cause was that the visual-mode mapping triggered the `Sel` and `Submit` functions once per selected line rather than once for the entire selection. The fix was ensuring the mapping operated on the selection as a single unit using the `\|` command separator.

Things to Watch Out For

1. **`term_list()[0]` assumes one terminal.** If multiple terminal buffers are open, the plugin sends code to the first one, which may not be the R session. Close unused terminals or modify the plugin to target a specific buffer.
2. **The `source_visual` temp file persists.** The plugin writes a temporary file to the current directory on every visual submission. Add `source_visual` to your `.gitignore` to prevent accidental commits.
3. **Long selections may cause timing issues.** The two-stage approach (write to file, then source) avoids most timing problems, but very large selections may still encounter race conditions between file writing and sourcing.
4. **`<CR>` remapping overrides Enter in normal mode.** This is intentional for R files but may surprise those who expect Enter to move the cursor down. The mapping only applies to R, Rmd, and Qmd filetypes.

Figure 3: Reference documentation for VimScript functions beside an open terminal editor

A minimal plugin teaches more about Vim's internals than a feature-rich one.

Lessons Learned

Conceptual Understanding

- Vim's terminal API (`term_sendkeys`, `term_list`) provides sufficient infrastructure for a complete REPL workflow without external dependencies.
- The two-stage submission pattern (write to file, then source) is more robust than sending text directly through `term_sendkeys` for multi-line selections.
- Scoping mappings to specific filetypes via autocommand groups prevents key conflicts in non-R buffers.
- A plugin under 150 lines can cover the vast majority of daily R interaction needs.

Technical Skills

- `getpos("'<")` and `getpos("'>")` extract the exact positions of visual selection boundaries.
- `expand("<cword>")` captures the word under the cursor, enabling generic inspection commands.
- `autocmd FileType` is the correct mechanism for filetype-scoped mappings; `ftplugin/` directories are an alternative but require more file management.
- The `\|` separator chains Vim commands on a single autocommand line, which is necessary for multi-step visual mode operations.

Gotchas and Pitfalls

- Visual mode mappings execute once per line by default unless the mapping explicitly operates on the selection as a unit.
- `term_list()` returns terminals in creation order, not in order of last use; the first terminal may not be the intended one.
- The `<C-r>%` expansion in mappings inserts the current filename, but fails if the filename contains spaces.
- `noh` after search-based navigation is necessary to avoid persistent highlighting that obscures the editing buffer.

Limitations

- The plugin targets Vim 8+ only; Neovim users would need to adapt the terminal API calls to Neovim's `jobsend()` and related functions.
- Only the first terminal buffer receives code submissions; multiple R sessions are not supported.
- The `source_visual` temporary file approach leaves artifacts in the working directory.
- No support for sending code to non-R interpreters (Python, Julia, etc.) without modification.
- The plugin does not provide R object completion, help lookup, or any of the advanced features that Nvim-R offers.
- Chunk navigation relies on markdown code fences and will not work in plain `.R` files.

Opportunities for Improvement

1. Add a function to identify the R terminal specifically (by checking the terminal command) rather than relying on `term_list()[0]`.
2. Use `tempname()` instead of a fixed filename for the temporary source file, preventing conflicts in multi-instance Vim sessions.
3. Extend the autocommand group to support Python and Julia filetypes with appropriate interpreter commands.
4. Add a function to display R help pages in a Vim buffer (e.g., `<localleader>?` on a function name).
5. Implement a chunk execution function that automatically advances to the next chunk after submission, matching the RStudio workflow.
6. Package the plugin as a proper Vim package with documentation in `doc/` for `:help` integration.

Every item on this list was later implemented. The plugin was renamed `zzvim-R`, packaged with a `doc/` help file, and extended with multi-terminal support, chunk-execution advance, and R object completion, among other additions well beyond this original scope. See Introducing zzvim-R for what the plugin looks like today.

Wrapping Up

Writing a Vim plugin for REPL interaction proved to teach more about Vim's terminal API than months of using other developers' plugins. The result (`rgt-R.vim`) is under 150 lines, has no external dependencies, and handles the core workflow of sending R code from an editing buffer to a running terminal session.

What this project made clear is that minimalism in tooling is undervalued. The established R-Vim plugins are powerful, but their complexity creates a barrier to understanding and customization. A small, readable plugin that one has written is easier to debug, extend, and teach from than a large plugin maintained by someone else.

Vim users working in R who find the existing plugins more complex than needed are encouraged to try writing their own. The terminal API is well-documented, and a working plugin can be built in an afternoon.

In conclusion, four points merit emphasis. First, Vim's `term_sendkeys()` and `term_list()` provide the complete infrastructure for REPL interaction without any external dependencies. Second, the two-stage submission pattern (write to file, then source) handles multi-line selections robustly, avoiding the timing issues of direct text injection. Third, autocommand groups scope mappings to relevant filetypes without affecting other buffers, keeping the plugin non-intrusive. Fourth, a minimal plugin under 150 lines covers 90% of daily R interaction needs, which is a favorable ratio of complexity to utility.

See Also

Related posts:

- Unix Command-Line Workspace Setup for Data Science: Terminal and shell configuration for development
- Introducing zzvim-R: The current, publicly released successor to the plugin documented here

Key resources:

- zzvim-R: The plugin's current GitHub repository
- Vim Terminal API: Official documentation for terminal functions
- Chris Toomey's Vim Talk: Inspiration for the plugin approach
- Nvim-R: The full-featured alternative for Neovim
- vim-slime: The tmux-based alternative
- Learn Vimscript the Hard Way: Comprehensive guide to writing Vim plugins

Reproducibility

The complete plugin source is presented inline above. To install:

1. Copy the code into `~/.vim/plugin/rgt-R.vim`.
2. Ensure your `localleader` is set (e.g., `let maplocalleader = ","` in `.vimrc`).
3. Open an R, Rmd, or Qmd file in Vim.
4. Press `<localleader>r` to open an R terminal.
5. Press `<CR>` on any line to submit it.

Project files:

```
simplevimplugin/
  analysis/report/index.qmd  (this post)
  plugin/rgt-R.vim            (the plugin)
  analysis/media/images/      (hero, ambiance)
```

Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from readers who:

- Spot an error or a better approach to any of the code in this post.
- Have suggestions for topics to cover.
- Want to discuss R programming, data science, or reproducible research.
- Have questions about anything in this tutorial.
- Simply want to say hello and connect.

Related posts in this cluster

This post is part of the *R Package Development and Testing* series. Recommended reading order:

1. Updating an R Package: A Complete Development Workflow
2. **Writing a Simple Vim Plugin for REPL Interaction** (this post)
3. Introducing zzvim-R
4. Testing Data Analysis Workflows in R
5. From testthat to tinytest: Converting an R Package Test Suite
6. Code Review for AI-Assisted R Package Development
7. Exemplar Code Review: The zzobj2fig Package