

# with-deepseek

in context neovim, help me create several custom lua function but mainly written with vimscript "vim.cmd", the function purpose to enhance coding experience

awesome, try to discover more custom lua function with vimscript inside

what about additional function we still not talk about in both previous responses

---

I'll help you create several custom Lua functions that leverage Vimscript to enhance your coding experience in NeoVim. These functions will bridge the power of both worlds.

## 1. Smart Comment Toggle Function

```
-- Toggle comments intelligently based on file type
function _G.smart_comment_toggle()
    local filetype = vim.bo.filetype
    local comment_string = vim.api.nvim_buf_get_option(0, 'commentstring')

    -- Use Vimscript to get comment format
    vim.cmd([[
        function! GetCommentFormat() abort
            let l:comment_format = &commentstring
            if l:comment_format == '' || l:comment_format == '%s'
                " Fallback comment formats by filetype
                if &filetype == 'lua'
                    return '-- %s'
                elseif &filetype == 'python' || &filetype == 'sh' || &filetype ==
'ruby'
                    return '# %s'
                elseif &filetype == 'vim'
                    return '" %s'
                else
                    return '/* %s */'
                endif
            endif
            return l:comment_format
        endfunction
    ]])

    local comment_format = vim.fn.GetCommentFormat()

    -- Extract left and right parts of comment
    local left, right = comment_format:match('^(.*)%s(.*?)$')
    left = left or ''
    right = right or ''

    -- Get current line
    local line = vim.fn.getline('.')

    -- Check if line is already commented
```

```

local is_commented = false
if left ~= '' then
    is_commented = line:match('^%s*' .. vim.pesc(left:gsub('%s', '')))
end

if is_commented then
    -- Uncomment the line
    local uncommented_line = line:gsub('%s*' .. vim.pesc(left:gsub('%s', '')),
    '')
    uncommented_line = uncommented_line:gsub(vim.pesc(right:gsub('%s', '')) ..
    '$', '')
    vim.fn.setline('.', uncommented_line)
else
    -- Comment the line
    local indent = line:match('^(%s*)')
    local commented_line = indent .. left:gsub('%s', '') .. line:gsub('%s*',
    '') .. right:gsub('%s', '')
    vim.fn.setline('.', commented_line)
end
end
end

```

## 2. Enhanced Buffer Management

```

function _G.smart_buffer_switch(direction)
    direction = direction or "next"

    vim.cmd([[
        function! HasModifiableBuffer() abort
            let l:bufs = filter(range(1, bufnr('$')), 'buflisted(v:val) &&
getbufvar(v:val, "&modifiable")')
            return len(l:bufs) > 1
        endfunction
    ]])

    local has_modifiable = vim.fn.HasModifiableBuffer()

    if has_modifiable == 1 then
        if direction == "next" then
            vim.cmd("bnext")
        else
            vim.cmd("bprevious")
        end
    else
        print("No other modifiable buffers available")
    end
end

-- Close buffer without closing window
function _G.smart_buffer_close()
    vim.cmd([[

```

```

function! CloseBufferKeepWindow() abort
    if len(filter(range(1, bufnr('$')), 'buflisted(v:val)')) > 1
        bdelete
    else
        echo "Last buffer - cannot close"
    endif
endfunction
]])

vim.fn.CloseBufferKeepWindow()
end

```

### 3. Advanced Search and Replace

```

function _G.visual_search_replace()
    local mode = vim.fn.mode()

    if mode ~= "v" and mode ~= "V" then
        print("Not in visual mode!")
        return
    end

    vim.cmd([[
        function! GetVisualSelection() abort
            let [line_start, column_start] = getpos("<")[1:2]
            let [line_end, column_end] = getpos(">")[1:2]
            let lines = getline(line_start, line_end)
            if len(lines) == 0
                return ''
            endif
            let lines[-1] = lines[-1][: column_end - 2]
            let lines[0] = lines[0][column_start - 1:]
            return join(lines, "\n")
        endfunction
    ]])

    local visual_selection = vim.fn.GetVisualSelection()

    if visual_selection == "" then
        print("No text selected!")
        return
    end

    -- Escape for Lua pattern and Vim regex
    local escaped_search = vim.fn.escape(visual_selection, '\\/. *$^~[]')
    local lua_escaped = visual_selection:gsub('([%(%)%.%%+%-%*%?%^%$%])',
'%%%1')

    -- Get replacement text
    local replace_with = vim.fn.input("Replace with: ")

```

```

    if replace_with ~= "" then
        -- Use Vimscript for the actual substitution (more reliable with complex
patterns)
        vim.cmd('::s/' .. escaped_search .. '/' .. vim.fn.escape(replace_with,
'\\') .. '/g')
        print("Replaced all occurrences")
    end
end
end

```

## 4. Code Formatting Helper

```

function _G.auto_format_code()
    local filetype = vim.bo.filetype

    vim.cmd([[
        function! HasFormatter() abort
            if &filetype == 'lua'
                return executable('stylua')
            elseif &filetype == 'python'
                return executable('black') || executable('autopep8')
            elseif &filetype == 'javascript' || &filetype == 'typescript'
                return executable('prettier')
            elseif &filetype == 'go'
                return 1 " gofmt is built-in
            else
                return 0
            endif
        endfunction
    ]])

    local has_formatter = vim.fn.HasFormatter()

    if has_formatter == 1 then
        -- Try LSP formatting first
        local success, _ = pcall(vim.lsp.buf.format)
        if not success then
            -- Fallback to filetype-specific formatting
            if filetype == "lua" then
                vim.cmd("!stylua %")
            elseif filetype == "python" then
                vim.cmd("!black %")
            elseif filetype == "javascript" or filetype == "typescript" then
                vim.cmd("!prettier --write %")
            elseif filetype == "go" then
                vim.cmd("!gofmt -w %")
            end
            vim.cmd("edit!") -- Reload the file
        end
        print("Formatted " .. filetype .. " code")
    end
end

```

```

else
    -- Use built-in formatting as last resort
    vim.cmd("normal! gg=G")
    print("Used basic formatting")
end
end
end

```

## 5. Smart Indentation Helper

```

function _G.fix_indentation()
    vim.cmd([[
        function! PreserveCursorPosition(command) abort
            let l:save = winsaveview()
            execute a:command
            call winrestview(l:save)
        endfunction
    ]])

    -- Save cursor position and execute commands
    vim.fn.PreserveCursorPosition("normal! mz")
    vim.fn.PreserveCursorPosition("execute 'normal! gg=G'")
    vim.fn.PreserveCursorPosition("normal! `z")

    print("Fixed indentation")
end

```

## 6. Enhanced Quickfix List Management

```

function _G.quickfix_helpers()
    local helpers = {}

    function helpers.filter_quickfix(pattern, invert)
        vim.cmd([[
            function! FilterQuickfix(pattern, invert) abort
                let l:new_list = []
                for l:item in getqflist()
                    let l:matches = l:item.text =~ a:pattern
                    if (a:invert && !l:matches) || (!a:invert && l:matches)
                        call add(l:new_list, l:item)
                    endif
                endfor
                call setqflist(l:new_list)
                echo "Filtered quickfix list"
            endfunction
        ]])

        vim.fn.FilterQuickfix(pattern, invert and 1 or 0)
    end

```

```

function helpers.export_quickfix()
    vim.cmd([[
        function! ExportQuickfix() abort
            let l:filename = input('Export to file: ', 'quickfix_export.txt')
            if l:filename == ''
                return
            endif
            execute 'redir > ' . l:filename
            silent execute 'copen'
            silent execute '%print'
            redir END
            echo 'Quickfix list exported to ' . l:filename
        endfunction
    ]])

    vim.fn.ExportQuickfix()
end

return helpers
end

```

## Usage Examples:

You can map these functions to keybindings:

```

-- Key mappings
vim.keymap.set('n', '<leader>cc', '<cmd>lua smart_comment_toggle(<CR>')
vim.keymap.set('n', '<leader>bn', '<cmd>lua smart_buffer_switch("next")<CR>')
vim.keymap.set('n', '<leader>bp', '<cmd>lua smart_buffer_switch("prev")<CR>')
vim.keymap.set('n', '<leader>bd', '<cmd>lua smart_buffer_close(<CR>')
vim.keymap.set('v', '<leader>sr', '<cmd>lua visual_search_replace(<CR>')
vim.keymap.set('n', '<leader>cf', '<cmd>lua auto_format_code(<CR>')
vim.keymap.set('n', '<leader>fi', '<cmd>lua fix_indentation(<CR>')

-- Quickfix helpers
local qf_helpers = quickfix_helpers()
vim.keymap.set('n', '<leader>qf', function()
    qf_helpers.filter_quickfix(vim.fn.input("Filter pattern: ")) end)
vim.keymap.set('n', '<leader>qe', function() qf_helpers.export_quickfix() end)

```

These functions combine the best of both Vimscript and Lua, using Vimscript for complex text manipulation and buffer operations where it excels, while leveraging Lua for better performance and integration with NeoVim's modern features.

---

Absolutely! Let me create more advanced custom Lua functions with Vimscript inside to further enhance your coding experience in NeoVim.

## 7. Advanced Git Integration

```

function _G.git_helpers()
    local helpers = {}

    function helpers.smart_git_blame()
        vim.cmd([[
            function! GetGitBlameInfo() abort
                if !executable('git')
                    return ['', 'Git not available']
                endif

                let l:file = expand('%:p')
                let l:line = line('.')
                let l:cmd = 'git blame -L ' . l:line . ',' . l:line . ' -- ' .
shellescape(l:file)
                let l:blame_output = systemlist(l:cmd)

                if v:shell_error != 0 || empty(l:blame_output)
                    return ['', 'Not in git repository or file not tracked']
                endif

                let l:blame_line = l:blame_output[0]
                let l:hash = matchstr(l:blame_line, '^\\^?\\zs\\x+')
                let l:author = matchstr(l:blame_line, '\\s*\\zs[^\\]+')
                let l:date = matchstr(l:blame_line, '\\d\\{4\\}-\\d\\{2\\}-\\d\\{2\\}')

                return [l:hash, trim(l:author) . ' (' . l:date . ')']
            endfunction
        ]])

        local hash, info = unpack(vim.fn.GetGitBlameInfo())

        if hash ~= "" then
            -- Create a floating window to show git info
            local buf = vim.api.nvim_create_buf(false, true)
            local width = 50
            local height = 3
            local row = 1
            local col = vim.o.columns - width - 1

            local opts = {
                style = "minimal",
                relative = "editor",
                width = width,
                height = height,
                row = row,
                col = col,
                border = "rounded"
            }

            local win = vim.api.nvim_open_win(buf, true, opts)

```

```

local lines = {
    "Git Blame Info:",
    "Commit: " .. hash:sub(1, 8),
    "Author: " .. info
}

vim.api.nvim_buf_set_lines(buf, 0, -1, false, lines)
vim.api.nvim_buf_set_option(buf, 'filetype', 'gitblame')
vim.api.nvim_buf_set_option(buf, 'modifiable', false)

-- Auto-close after 3 seconds
vim.defer_fn(function()
    if vim.api.nvim_win_is_valid(win) then
        vim.api.nvim_win_close(win, true)
    end
end, 3000)
else
    print(info)
end
end

function helpers.git_diff_current_line()
    vim.cmd([[
        function! GetGitDiffForLine() abort
            if !executable('git')
                return ''
            endif

            let l:file = expand('%:p')
            let l:line = line('.')
            let l:cmd = 'git diff -U0 -- ' . shellescape(l:file) . ' | grep -A3
-B3 "^@@.*+' . l:line . ''
            let l:diff_output = system(l:cmd)

            return split(l:diff_output, "\n")
        endfunction
    ]])

    local diff_lines = vim.fn.GetGitDiffForLine()

    if #diff_lines > 0 then
        -- Create scratch buffer with diff
        vim.cmd('botright 10new')
        local buf = vim.api.nvim_get_current_buf()
        vim.api.nvim_buf_set_name(buf, 'git_diff_line')
        vim.api.nvim_buf_set_lines(buf, 0, -1, false, diff_lines)
        vim.api.nvim_buf_set_option(buf, 'modifiable', false)
        vim.api.nvim_buf_set_option(buf, 'buftype', 'nofile')
        vim.api.nvim_buf_set_option(buf, 'filetype', 'diff')

        vim.cmd('setlocal wrap')
        print("Showing git diff for current line")
    end
end

```



```

        else
            print("No changes for current line")
        end
    end

    return helpers
end

```

## 8. Advanced Code Navigation

```

function _G.enhanced_navigation()
    local nav = {}

    function nav.jump_to_related_file()
        vim.cmd([[
            function! FindRelatedFiles() abort
                let l:current_file = expand('%:t:r')
                let l:extension = expand('%:e')
                let l:dir = expand('%:p:h')
                let l:patterns = []

                " Common related file patterns
                if &filetype == 'javascript' || &filetype == 'typescript'
                    call add(l:patterns, l:current_file . '.test.' . l:extension)
                    call add(l:patterns, l:current_file . '.spec.' . l:extension)
                    call add(l:patterns, l:current_file . '.styles.' . l:extension)
                    call add(l:patterns, l:current_file . '.types.' . l:extension)
                elseif &filetype == 'python'
                    call add(l:patterns, l:current_file . '_test.py')
                    call add(l:patterns, l:current_file . '_test.py')
                elseif &filetype == 'cpp' || &filetype == 'c'
                    call add(l:patterns, l:current_file . '.h')
                    call add(l:patterns, l:current_file . '.hpp')
                endif

                " Also check for same name with different extensions
                for l:ext in ['js', 'ts', 'jsx', 'tsx', 'css', 'scss', 'html']
                    if l:ext != l:extension
                        call add(l:patterns, l:current_file . '.' . l:ext)
                    endif
                endfor

                return l:patterns
            endfunction
        ]])

        local patterns = vim.fn.FindRelatedFiles()

        for _, pattern in ipairs(patterns) do
            local found = vim.fn.glob(pattern)

```

```

        if found ~= "" then
            vim.cmd('edit ' .. found)
            return
        end
    end
end

print("No related files found")
end

function nav.smart_jump_to_definition()
    vim.cmd([[
        function! TryLspDefinition() abort
            try
                call v:lua.vim.lsp.buf.definition()
                return 1
            catch
                return 0
            endtry
        endfunction

        function! FallbackJump() abort
            " Try ctags
            if executable('ctags')
                let l:tag = expand('<cword>')
                let l:result = system('ctags -x --language-force=' . &filetype .
' ' . shellescape(expand('%'))) . ' | grep -w "' . l:tag . '"')
                if !empty(l:result)
                    execute 'tag ' . l:tag
                    return 1
                endif
            endif

            " Try grep in current directory
            let l:search = expand('<cword>')
            execute 'vimgrep /' . l:search . '/j **/*.' . expand('%:e')
            if len(getqflist()) > 0
                copen
                return 1
            endif

            return 0
        endfunction
    ]])

    local lsp_success = vim.fn.TryLspDefinition()
    if lsp_success == 0 then
        local fallback_success = vim.fn.FallbackJump()
        if fallback_success == 0 then
            print("No definition found")
        end
    end
end
end

```

```
return nav
end
```

## 9. Advanced Text Manipulation

```
function _G.advanced_text_operations()
    local text_ops = {}

    function text_ops.smart_case_conversion()
        vim.cmd([[
            function! AnalyzeTextCase(text) abort
                if a:text =~# '^[A-Z][A-Z_0-9]*$'
                    return 'SCREAMING_SNAKE'
                elseif a:text =~# '^[a-z][a-z0-9]*_[a-z]'
                    return 'snake_case'
                elseif a:text =~# '^[a-z][a-zA-Z0-9]*[A-Z]'
                    return 'camelCase'
                elseif a:text =~# '^[A-Z][a-zA-Z0-9]*[a-z]'
                    return 'PascalCase'
                elseif a:text =~# '^[a-z][a-z0-9]*-[a-z]'
                    return 'kebab-case'
                else
                    return 'unknown'
                endif
            endfunction

            function! ConvertCase(text, target_case) abort
                let l:text = a:text

                if a:target_case == 'snake_case'
                    let l:text = tolower(l:text)
                    let l:text = substitute(l:text, '\([A-Z]\)', '_\l\1', 'g')
                    let l:text = substitute(l:text, '-', '_', 'g')
                    let l:text = substitute(l:text, '__', '_', 'g')
                    return l:text
                elseif a:target_case == 'SCREAMING_SNAKE'
                    let l:text = toupper(l:text)
                    let l:text = substitute(l:text, '\([a-z]\)', '_\U\1', 'g')
                    let l:text = substitute(l:text, '-', '_', 'g')
                    let l:text = substitute(l:text, '__', '_', 'g')
                    return l:text
                elseif a:target_case == 'camelCase'
                    let l:text = tolower(l:text)
                    let l:text = substitute(l:text, '[_-]\(\w\)', '\u\1', 'g')
                    return l:text
                elseif a:target_case == 'PascalCase'
                    let l:text = substitute(l:text, '^.', '\u&', '')
                    let l:text = substitute(l:text, '[_-]\(\w\)', '\u\1', 'g')
                    return l:text
                endif
            endfunction
        ]])
    endfunction
endfunction
```

```

        elseif a:target_case == 'kebab-case'
            let l:text = tolower(l:text)
            let l:text = substitute(l:text, '_', '-', 'g')
            let l:text = substitute(l:text, '\([A-Z]\)', '-\l\1', 'g')
            return l:text
        endif

        return a:text
    endfunction
]])

local visual_mode = vim.fn.mode()
local text = ""

if visual_mode == "v" or visual_mode == "V" then
    text = vim.fn.GetVisualSelection()
else
    text = vim.fn.expand('<cword>')
end

if text == "" then
    print("No text selected")
    return
end

local current_case = vim.fn.AnalyzeTextCase(text)
print("Current case: " .. current_case)

local choices = {
    "1. snake_case",
    "2. SCREAMING_SNAKE",
    "3. camelCase",
    "4. PascalCase",
    "5. kebab-case"
}

print("Convert to:")
for _, choice in ipairs(choices) do
    print(choice)
end

local choice = vim.fn.input("Choose (1-5): ")
local target_cases = {
    ["1"] = "snake_case",
    ["2"] = "SCREAMING_SNAKE",
    ["3"] = "camelCase",
    ["4"] = "PascalCase",
    ["5"] = "kebab-case"
}

local target_case = target_cases[choice]
if target_case then

```

```

    local converted = vim.fn.ConvertCase(text, target_case)

    if visual_mode == "v" or visual_mode == "V" then
        vim.api.nvim_feedkeys(vim.api.nvim_replace_termcodes("<Esc>", true,
false, true), 'n', true)
        vim.fn.setline('.', converted)
    else
        vim.fn.setline('.', vim.fn.getline('.'):gsub(vim.pesc(text),
converted, 1))
    end

    print("Converted to: " .. target_case)
else
    print("Invalid choice")
end
end

function text_ops.align_text(separator)
    separator = separator or "="

    vim.cmd([[
        function! AlignLines(sep) abort
            let l:lines = getline("<", ">")
            let l:max_pos = 0

            " Find maximum separator position
            for l:line in l:lines
                let l:pos = stridx(l:line, a:sep)
                if l:pos > l:max_pos
                    let l:max_pos = l:pos
                endif
            endfor

            " Align all lines
            let l:new_lines = []
            for l:line in l:lines
                let l:pos = stridx(l:line, a:sep)
                if l:pos > 0
                    let l:spaces_needed = l:max_pos - l:pos
                    let l:new_line = l:line[0:l:pos-1] . repeat(' ',
l:spaces_needed) . l:line[l:pos:]
                    call add(l:new_lines, l:new_line)
                else
                    call add(l:new_lines, l:line)
                endif
            endfor

            return l:new_lines
        endfunction
    ]])

    local mode = vim.fn.mode()

```

```

if mode ~= "v" and mode ~= "V" then
    print("Not in visual mode!")
    return
end

local aligned_lines = vim.fn.AlignLines(separator)
vim.api.nvim_feedkeys(vim.api.nvim_replace_termcodes("<Esc>", true, false,
true), 'n', true)

local start_line = vim.fn.line("'<")
local end_line = vim.fn.line("'>")

vim.api.nvim_buf_set_lines(0, start_line - 1, end_line, false,
aligned_lines)
print("Aligned text around '" .. separator .. "'")
end

return text_ops
end

```

## 10. Project Management Helpers

```

function _G.project_helpers()
    local project = {}

    function project.detect_project_type()
        vim.cmd([[
            function! DetectProjectType() abort
                let l:files = split(glob('*'), "\n")
                let l:indicators = {
                    \ 'nodejs': ['package.json', 'node_modules'],
                    \ 'python': ['requirements.txt', 'setup.py', 'pyproject.toml'],
                    \ 'rust': ['Cargo.toml'],
                    \ 'go': ['go.mod'],
                    \ 'ruby': ['Gemfile'],
                    \ 'php': ['composer.json'],
                    \ 'java': ['pom.xml', 'build.gradle']
                    \ }

                for [l:type, l:markers] in items(l:indicators)
                    for l:marker in l:markers
                        if index(l:files, l:marker) != -1
                            return l:type
                        endif
                    endfor
                endfor

                return 'unknown'
            endfunction
        ]])
    end
end

```

```

local project_type = vim.fn.DetectProjectType()

-- Set project-specific settings
if project_type == "nodejs" then
    vim.bo.tabstop = 2
    vim.bo.shiftwidth = 2
    print("Detected: Node.js project (set tabstop=2)")
elseif project_type == "python" then
    vim.bo.tabstop = 4
    vim.bo.shiftwidth = 4
    print("Detected: Python project (set tabstop=4)")
elseif project_type == "go" then
    vim.bo.tabstop = 4
    vim.bo.shiftwidth = 4
    print("Detected: Go project")
else
    print("Project type: " .. project_type)
end

return project_type
end

function project.toggle_project_tree()
    vim.cmd([[
        function! ToggleProjectTree() abort
            if exists('b:project_tree_open') && b:project_tree_open
                if exists('b:project_tree_bufnr') &&
bufexists(b:project_tree_bufnr)
                    execute 'bwipeout ' . b:project_tree_bufnr
                endif
                let b:project_tree_open = 0
                return 0
            else
                let b:project_tree_open = 1
                return 1
            endif
        endfunction
    ]])

    local should_open = vim.fn.ToggleProjectTree()

    if should_open == 1 then
        -- Create project tree in vertical split
        vim.cmd('vnew')
        local buf = vim.api.nvim_get_current_buf()
        vim.api.nvim_buf_set_name(buf, 'Project Tree')
        vim.api.nvim_buf_set_option(buf, 'buftype', 'nofile')
        vim.api.nvim_buf_set_option(buf, 'modifiable', true)

        -- Get directory structure
        local files = vim.fn.split(vim.fn.glob('**/*'), "\n")
    end
end

```

```

local tree_lines = {'.'}

for _, file in ipairs(files) do
    local depth = select(2, string.gsub(file, '/', ''))
    local indent = string.rep(' ', depth)
    local display_name = file
    tree_lines[#tree_lines + 1] = indent .. '📄 ' .. display_name
end

vim.api.nvim_buf_set_lines(buf, 0, -1, false, tree_lines)
vim.api.nvim_buf_set_option(buf, 'modifiable', false)

-- Set buffer-local variable
vim.b.project_tree_bufnr = buf

-- Add keymaps for the project tree
vim.keymap.set('n', '<CR>', function()
    local line = vim.fn.getline('.')
    local file = line:gsub('^s*📄%s*', ''):gsub('^%s+', '')
    if file ~= '.' then
        vim.cmd('edit ' .. file)
    end
end, {buffer = buf})

print("Project tree opened")
else
    print("Project tree closed")
end
end

return project
end

```

## 11. Session Management

```

function _G.session_management()
    local session = {}

    function session.save_named_session()
        vim.cmd([[
            function! GetSessionName() abort
                let l:default_name = fnamemodify(getcwd(), ':t')
                let l:name = input('Session name: ', l:default_name)
                if l:name == ''
                    return ''
                endif
                return '~/.local/share/nvim/sessions/' . l:name . '.vim'
            endfunction
        ]])
    end
end

```



```

local session_file = vim.fn.GetSessionName()
if session_file ~= "" then
    vim.cmd('mksession! ' .. session_file)
    print("Session saved: " .. session_file)
end
end

function session.load_session()
    vim.cmd([[
        function! ListSessions() abort
            let l:session_dir = '~/.local/share/nvim/sessions'
            let l:session_files = split(glob(l:session_dir . '/*.vim'), "\n")
            let l:sessions = []
            for l:file in l:session_files
                call add(l:sessions, fnamemodify(l:file, ':t:r'))
            endfor
            return l:sessions
        endfunction
    ]])

    local sessions = vim.fn.ListSessions()

    if #sessions == 0 then
        print("No saved sessions found")
        return
    end

    print("Available sessions:")
    for i, name in ipairs(sessions) do
        print(i .. ". " .. name)
    end

    local choice = vim.fn.input("Choose session (number): ")
    local session_num = tonumber(choice)

    if session_num and session_num > 0 and session_num <= #sessions then
        local session_file = '~/.local/share/nvim/sessions/' ..
sessions[session_num] .. '.vim'
        vim.cmd('source ' .. session_file)
        print("Loaded session: " .. sessions[session_num])
    else
        print("Invalid choice")
    end
end

return session
end

```

## Usage Mappings:

```

-- Git helpers
local git = git_helpers()
vim.keymap.set('n', '<leader>gb', git.smart_git_blame)
vim.keymap.set('n', '<leader>gd', git.git_diff_current_line)

-- Navigation
local nav = enhanced_navigation()
vim.keymap.set('n', '<leader>rf', nav.jump_to_related_file)
vim.keymap.set('n', 'gd', nav.smart_jump_to_definition)

-- Text operations
local text_ops = advanced_text_operations()
vim.keymap.set('n', '<leader>sc', text_ops.smart_case_conversion)
vim.keymap.set('v', '<leader>sc', text_ops.smart_case_conversion)
vim.keymap.set('v', '<leader>al', function() text_ops.align_text("=") end)

-- Project management
local project = project_helpers()
vim.keymap.set('n', '<leader>pt', project.detect_project_type)
vim.keymap.set('n', '<leader>pt', project.toggle_project_tree)

-- Session management
local session = session_management()
vim.keymap.set('n', '<leader>ss', session.save_named_session)
vim.keymap.set('n', '<leader>sl', session.load_session)

```

These advanced functions provide powerful enhancements for Git integration, code navigation, text manipulation, project management, and session handling - all while leveraging the strengths of both Vimscript and Lua!

---

Excellent! Let me create even more unique and powerful functions that we haven't covered yet. These will focus on advanced workflows, debugging, documentation, and productivity enhancements.

## 12. Advanced Debugging Helpers

```

function _G.debugging_helpers()
    local debug = {}

    function debug.toggle_debug_statements()
        vim.cmd([[
            function! GetDebugStatements() abort
                let l:debug_patterns = {
                    \ 'python': ['print(', 'import pdb', 'breakpoint()',
'pdb.set_trace()'],
                    \ 'javascript': ['console.log', 'debugger', 'console.error'],
                    \ 'lua': ['print(', 'vim.pretty_print', 'require("debug")'],
                    \ 'cpp': ['printf(', 'cout <<', 'std::cout', '// DEBUG'],
                    \ 'java': ['System.out.println', 'Log.d', 'Log.e'],
                    \ 'php': ['var_dump(', 'print_r(', 'error_log(']
                \ }

```

```

        let l:ft = &filetype
        if has_key(l:debug_patterns, l:ft)
            return l:debug_patterns[l:ft]
        endif
        return []
    endfunction
endfunction
]])

local debug_patterns = vim.fn.GetDebugStatements()
local current_line = vim.fn.getline('.')

-- Check if current line contains debug statement
local is_debug_line = false
for _, pattern in ipairs(debug_patterns) do
    if current_line:find(vim.pesc(pattern), 1, true) then
        is_debug_line = true
        break
    end
end

if is_debug_line then
    -- Comment out debug line
    vim.cmd('normal! gcc')
else
    -- Insert appropriate debug statement
    local filetype = vim.bo.filetype
    local debug_template = ""

    if filetype == "python" then
        debug_template = "print(f\"DEBUG: { }\")"
    elseif filetype == "javascript" or filetype == "typescript" then
        debug_template = "console.log('DEBUG:', );"
    elseif filetype == "lua" then
        debug_template = "print('DEBUG: ')"
    elseif filetype == "cpp" then
        debug_template = "std::cout << \"DEBUG: \" << << std::endl;"
    else
        debug_template = "// DEBUG: "
    end

    vim.fn.append('.', debug_template)
    vim.cmd('normal! j0f)hi') -- Position cursor inside parentheses
end
end

function debug.visual_debug_wrapper()
    local mode = vim.fn.mode()
    if mode ~= "v" and mode ~= "V" then
        print("Not in visual mode!")
        return
    end
end

```

```

local selected_text = vim.fn.GetVisualSelection()
if selected_text == "" then
    return
end

vim.cmd([[
    function! CreateDebugWrapper(text, filetype) abort
        let l:wrappers = {
            \ 'python': 'print(f"DEBUG ' . a:filetype . ': {}".format(' .
a:text . '))',
            \ 'javascript': 'console.log("DEBUG ' . a:filetype . ': ", ' .
a:text . ');',
            \ 'lua': 'print("DEBUG ' . a:filetype . ': " .. tostring(' .
a:text . '))',
            \ 'cpp': 'std::cout << "DEBUG ' . a:filetype . ': " << ' .
a:text . ' << std::endl;'
            \ }

        if has_key(l:wrappers, a:filetype)
            return l:wrappers[a:filetype]
        else
            return '/* DEBUG: ' . a:text . ' */'
        endif
    endfunction
]])

local filetype = vim.bo.filetype
local debug_line = vim.fn.CreateDebugWrapper(selected_text, filetype)

-- Replace visual selection with debug wrapper
vim.api.nvim_feedkeys(vim.api.nvim_replace_termcodes("<Esc>", true, false,
true), 'n', true)
vim.fn.setline('.', debug_line)

print("Wrapped selection in debug statement")
end

return debug
end

```

## 13. Advanced Documentation Generators

```

function _G.documentation_helpers()
    local docs = {}

    function docs.generate_function_docs()
        vim.cmd([[
            function! ExtractFunctionSignature() abort
                let l:line = line('.')

```

```

let l:filetype = &filetype

if l:filetype == 'python'
    " Find function definition
    let l:func_line = search('^def\s', 'bnW')
    if l:func_line
        return getline(l:func_line)
    endif
elseif l:filetype == 'javascript' || l:filetype == 'typescript'
    let l:func_line = search('^function\s\|^const\s.*=\s*'
(\|^async\s', 'bnW')
    if l:func_line
        return getline(l:func_line)
    endif
elseif l:filetype == 'lua'
    let l:func_line = search('^function\s', 'bnW')
    if l:func_line
        return getline(l:func_line)
    endif
endif

return ''
endfunction

function! ParseParameters(func_line, filetype) abort
    let l:params = []

    if a:filetype == 'python'
        let l:match = matchstr(a:func_line, 'def\s\+\w\+\s*(\
([^\)]*\))')
        if !empty(l:match)
            let l:param_str = matchstr(l:match, '(\zs[^\)]*\ze)')
            let l:params = split(l:param_str, ',')
        endif
    elseif a:filetype == 'javascript' || a:filetype == 'typescript'
        let l:match = matchstr(a:func_line, 'function\s\+\w\+\s*(\
([^\)]*\))\|=\\s*(\\s*\\([^\)]*\))\\s*=>')
        if !empty(l:match)
            let l:param_str = matchstr(l:match, '(\zs[^\)]*\ze)')
            let l:params = split(l:param_str, ',')
        endif
    elseif a:filetype == 'lua'
        let l:match = matchstr(a:func_line, 'function\s\+\w\+\s*(\
([^\)]*\))')
        if !empty(l:match)
            let l:param_str = matchstr(l:match, '(\zs[^\)]*\ze)')
            let l:params = split(l:param_str, ',')
        endif
    endif

    " Clean up parameters
    return map(l:params, 'trim(v:val)')
endfunction

```

```

        endfunction
    end)

    local func_line = vim.fn.ExtractFunctionSignature()
    if func_line == "" then
        print("No function definition found above cursor")
        return
    end

    local filetype = vim.bo.filetype
    local params = vim.fn.ParseParameters(func_line, filetype)

    -- Generate documentation based on filetype
    local doc_lines = {}
    local func_name = vim.fn.matchstr(func_line, '\\
(def\\|function\\)\\s\\+\\zs\\w\\+\\ze')

    if filetype == "python" then
        table.insert(doc_lines, '"""')
        table.insert(doc_lines, func_name .. ' - TODO: Add description')
        table.insert(doc_lines, '')
        if #params > 0 then
            table.insert(doc_lines, 'Args:')
            for _, param in ipairs(params) do
                if param ~= "self" then
                    table.insert(doc_lines, '    ' .. param .. ': TODO: Add
description')
                end
            end
        end
        table.insert(doc_lines, '')
        table.insert(doc_lines, 'Returns:')
        table.insert(doc_lines, '    TODO: Add return description')
        table.insert(doc_lines, '"""')
    elseif filetype == "javascript" or filetype == "typescript" then
        table.insert(doc_lines, '/*')
        table.insert(doc_lines, ' * ' .. func_name .. ' - TODO: Add
description')
        table.insert(doc_lines, ' *')
        if #params > 0 then
            for _, param in ipairs(params) do
                table.insert(doc_lines, ' * @param {' .. param .. '} TODO: Add
description')
            end
        end
        table.insert(doc_lines, ' * @returns {*} TODO: Add return description')
        table.insert(doc_lines, ' */')
    elseif filetype == "lua" then
        table.insert(doc_lines, '----' .. func_name .. ' - TODO: Add
description')
        if #params > 0 then
            for _, param in ipairs(params) do

```

```

        table.insert(doc_lines, '---@param ' .. param .. ' any TODO: Add
description')
    end
end
    table.insert(doc_lines, '---@return any TODO: Add return description')
else
    table.insert(doc_lines, '/*')
    table.insert(doc_lines, ' * ' .. func_name .. ' - TODO: Add
description')
    table.insert(doc_lines, ' */')
end

-- Insert documentation above function
local current_line = vim.fn.line('.')
vim.fn.append(current_line - 1, doc_lines)

print("Generated documentation template for " .. func_name)
end

function docs.create_todo_comment()
    vim.cmd([[
        function! GetTodoTemplate() abort
            let l:username = $USER
            if empty(l:username)
                let l:username = 'unknown'
            endif

            let l:date = strftime('%Y-%m-%d')
            return 'TODO(' . l:username . '): ' . l:date . ' - '
        endfunction
    ]])

    local todo_template = vim.fn.GetTodoTemplate()
    local comment_char = ""

    -- Get comment character for current filetype
    local comment_string = vim.bo.commentstring
    if comment_string and comment_string ~= "" then
        comment_char = comment_string:match("^(.*)%%s") or "/"
    else
        comment_char = "/"
    end

    local todo_line = comment_char .. " " .. todo_template
    vim.fn.append('.', {todo_line})
    vim.cmd('normal! j$') -- Move to end of line for typing

    print("Created TODO comment")
end

```

```

    return docs
end

```

## 14. Advanced Code Analysis

```

function _G.code_analysis_helpers()
    local analysis = {}

    function analysis.analyze_function_complexity()
        vim.cmd([[
            function! CalculateCyclomaticComplexity() abort
                let l:complexity = 0
                let l:start_line = line('.')
                let l:end_line = search('^\\s*end\\|\\s*}\\|\\s*$', 'nW')

                if l:end_line == 0
                    let l:end_line = line('$')
                endif

                " Count decision points
                for l:ln in range(l:start_line, l:end_line)
                    let l:line = getline(l:ln)
                    if l:line =~? 'if\\|else\\|case\\|for\\|while\\|&&\\||\\|\\|?
\\|catch'
                        let l:complexity += 1
                    endif
                endfor

                return l:complexity + 1 " Base complexity
            endfunction

            function! AnalyzeFunctionMetrics() abort
                let l:metrics = {}
                let l:start_line = line('.')
                let l:end_line = search('^\\s*end\\|\\s*}\\|\\s*$', 'nW')

                if l:end_line == 0
                    let l:end_line = line('$')
                endif

                let l:metrics.lines = l:end_line - l:start_line + 1
                let l:metrics.complexity = CalculateCyclomaticComplexity()

                " Count variables
                let l:metrics.variables = 0
                for l:ln in range(l:start_line, l:end_line)
                    let l:line = getline(l:ln)
                    if l:line =~? '\\<var\\s\\|\\|<let\\s\\|\\|<const\\s\\|=\\s*[^=]'
                        let l:metrics.variables += 1
                    endif
                endfor
            endfunction
        ]])
    endfunction
endfunction

```



```

        endfor

        return l:metrics
    endfunction
]])

local metrics = vim.fn.AnalyzeFunctionMetrics()

-- Create analysis report
local report = {
    "Function Analysis Report:",
    "Lines of code: " .. metrics.lines,
    "Cyclomatic complexity: " .. metrics.complexity,
    "Variables used: " .. metrics.variables,
    "",
    "Complexity Assessment:"
}

if metrics.complexity <= 5 then
    table.insert(report, "✅ Low complexity - Good")
elseif metrics.complexity <= 10 then
    table.insert(report, "⚠️ Moderate complexity - Consider refactoring")
else
    table.insert(report, "❌ High complexity - Strongly consider
refactoring")
end

-- Show in floating window
local buf = vim.api.nvim_create_buf(false, true)
local width = 40
local height = #report
local row = 1
local col = vim.o.columns - width - 1

local opts = {
    style = "minimal",
    relative = "editor",
    width = width,
    height = height,
    row = row,
    col = col,
    border = "rounded"
}

local win = vim.api.nvim_open_win(buf, true, opts)
vim.api.nvim_buf_set_lines(buf, 0, -1, false, report)
vim.api.nvim_buf_set_option(buf, 'modifiable', false)
vim.api.nvim_buf_set_option(buf, 'filetype', 'analysis')

-- Auto-close after 5 seconds
vim.defer_fn(function()
    if vim.api.nvim_win_is_valid(win) then

```

```

        vim.api.nvim_win_close(win, true)
    end
end, 5000)
end

function analysis.find_code_smells()
    vim.cmd([[
        function! DetectCodeSmells() abort
            let l:smells = []
            let l:filetype = &filetype

            " Long function detection
            let l:long_functions = []
            let l:line = 1
            while l:line <= line('$')
                if getline(l:line) =~? '^def\\s\\|\\^function\\s'
                    let l:func_start = l:line
                    let l:func_end = search('^\\s*end\\|\\^\\s*}\\|\\^$', 'nW')
                    if l:func_end == 0
                        let l:func_end = line('$')
                    endif

                    if (l:func_end - l:func_start) > 30
                        call add(l:long_functions, 'Long function: ' .
getline(l:func_start) . ' (lines: ' . (l:func_end - l:func_start) . ')')
                    endif
                    let l:line = l:func_end
                else
                    let l:line += 1
                endif
            endwhile

            " Deep nesting detection
            let l:deep_nesting = []
            let l:line = 1
            while l:line <= line('$')
                let l:line_text = getline(l:line)
                let l:indent = len(matchstr(l:line_text, '^\\s*'))
                if l:indent > 12 " More than 3 levels of indentation (assuming 4
spaces per level)
                    call add(l:deep_nesting, 'Deep nesting at line ' . l:line .
': ' . l:line_text)
                endif
                let l:line += 1
            endwhile

            " Magic numbers
            let l:magic_numbers = []
            let l:line = 1
            while l:line <= line('$')
                let l:line_text = getline(l:line)
                if l:line_text =~? '\\d\\(\\d\\|\\.\\)\\|[a-zA-Z0-9._]' &&

```

```

l:line_text !~? '0\\|1\\|2\\|100\\|255\\|1024'
    call add(l:magic_numbers, 'Magic number at line ' . l:line .
': ' . matchstr(l:line_text, '\\d\\(\\d\\|\\.\\)\\+'))
    endif
    let l:line += 1
endwhile

return {
    \ 'long_functions': l:long_functions,
    \ 'deep_nesting': l:deep_nesting,
    \ 'magic_numbers': l:magic_numbers
    \ }
endfunction
]])

local smells = vim.fn.DetectCodeSmells()

-- Create quickfix list with code smells
local qf_items = {}

for _, smell in ipairs(smells.long_functions) do
    table.insert(qf_items, {
        text = smell,
        type = 'W'
    })
end

for _, smell in ipairs(smells.deep_nesting) do
    local line_num = tonumber(smell:match('line (%d+)'))
    table.insert(qf_items, {
        filename = vim.fn.expand('%'),
        lnum = line_num,
        text = smell,
        type = 'W'
    })
end

for _, smell in ipairs(smells.magic_numbers) do
    local line_num = tonumber(smell:match('line (%d+)'))
    table.insert(qf_items, {
        filename = vim.fn.expand('%'),
        lnum = line_num,
        text = smell,
        type = 'I'
    })
end

vim.fn.setqflist(qf_items)
vim.cmd('copen')
print("Found " .. #qf_items .. " potential code smells")
end

```

```
    return analysis
end
```

## 15. Advanced Snippet Management

```
function _G.snippet_management()
    local snippets = {}

    function snippets.expand_custom_snippet()
        vim.cmd([[
            function! GetCustomSnippets() abort
                let l:snippets = {
                    \ 'fori': ['for (let i = 0; i < ; i++) {', '}', ''],
                    \ 'foreach': ['for (const item of ) {', '}', ''],
                    \ 'func': ['function () {', '}', ''],
                    \ 'if': ['if () {', '}', ''],
                    \ 'try': ['try {', '}' catch (error) {', '}', ''],
                    \ 'class': ['class {', 'constructor() {', '}', '}', ''],
                    \ }
                return l:snippets
            endfunction
        ]])

        local snippet_key = vim.fn.input("Snippet key: ")
        local custom_snippets = vim.fn.GetCustomSnippets()

        if custom_snippets[snippet_key] then
            local snippet_lines = custom_snippets[snippet_key]
            vim.fn.append('.', snippet_lines)

            -- Position cursor at first placeholder
            vim.cmd('normal! j0f)hi')
            print("Expanded snippet: " .. snippet_key)
        else
            print("Snippet not found: " .. snippet_key)
        end
    end

    function snippets.save_selection_as_snippet()
        local mode = vim.fn.mode()
        if mode =~ "v" and mode =~ "V" then
            print("Not in visual mode!")
            return
        end

        local selected_text = vim.fn.GetVisualSelection()
        if selected_text == "" then
            return
        end
    end
end
```

```

local snippet_name = vim.fn.input("Snippet name: ")
if snippet_name == "" then
    return
end

vim.cmd([[
    function! SaveSnippet(name, content) abort
        let l:snippet_file = '~/.config/nvim/snippets/' . a:name . '.snip'
        let l:dir = fnamemodify(l:snippet_file, ':h')

        " Create directory if it doesn't exist
        if !isdirectory(l:dir)
            call mkdir(l:dir, 'p')
        endif

        " Write snippet file
        call writefile(split(a:content, "\n"), expand(l:snippet_file))
        return l:snippet_file
    endfunction
]])

local snippet_file = vim.fn.SaveSnippet(snippet_name, selected_text)
print("Saved snippet to: " .. snippet_file)
end

return snippets
end

```

## 16. Advanced Window Management

```

function _G.advanced_window_management()
    local windows = {}

    function windows.balance_windows_smart()
        vim.cmd([[
            function! SmartWindowBalance() abort
                let l:total_windows = winnr('$')
                let l:current_layout = []

                " Analyze current window layout
                for l:win in range(1, l:total_windows)
                    call win_execute(l:win, 'let l:current_layout += [winwidth(0) .
"x" . winheight(0)]')
                endfor

                " Simple heuristic: if windows are very uneven, balance them
                let l:min_width = min(map(copy(l:current_layout), {_, v -> split(v,
"x")[0]}))

                let l:max_width = max(map(copy(l:current_layout), {_, v -> split(v,
"x")[0]}))

```

```

        if l:max_width - l:min_width > 20
            return 1 " Needs balancing
        else
            return 0 " Already balanced
        endif
    endfunction
endfunction
]])

local needs_balancing = vim.fn.SmartWindowBalance()

if needs_balancing == 1 then
    vim.cmd('wincmd =')
    print("Windows balanced")
else
    print("Windows already well balanced")
end
end

function windows.create_scratch_buffer()
    vim.cmd([[
        function! CreateScratchBuffer() abort
            let l:scratch_buf = bufadd('scratch://' . strftime('%H%M%S'))
            call bufload(l:scratch_buf)
            call setbufvar(l:scratch_buf, '&buftype', 'nofile')
            call setbufvar(l:scratch_buf, '&bufhidden', 'hide')
            call setbufvar(l:scratch_buf, '&swapfile', 0)
            return l:scratch_buf
        endfunction
    ]])

    local scratch_buf = vim.fn.CreateScratchBuffer()
    vim.cmd('sbuffer ' .. scratch_buf)

    -- Set some helpful text
    local header = {
        "SCRATCH BUFFER - " .. os.date("%Y-%m-%d %H:%M:%S"),
        "Use this for temporary notes, calculations, or experiments",
        "This buffer will not be saved automatically",
        "",
        "-----",
        ""
    }

    vim.api.nvim_buf_set_lines(scratch_buf, 0, -1, false, header)
    vim.api.nvim_buf_set_option(scratch_buf, 'modifiable', true)

    print("Created scratch buffer")
end

function windows.rotate_windows()
    vim.cmd([[

```

```

function! RotateWindowLayout() abort
    let l:current_win = winnr()
    let l:total_windows = winnr('$')

    if l:total_windows < 2
        echo "Need at least 2 windows to rotate"
        return
    endif

    " Get all window IDs
    let l:win_ids = []
    for l:i in range(1, l:total_windows)
        call add(l:win_ids, win_getid(l:i))
    endfor

    " Rotate the list
    call add(l:win_ids, remove(l:win_ids, 0))

    " Rearrange windows
    for l:i in range(1, l:total_windows)
        call win_gotoid(l:win_ids[l:i])
        execute (l:i + 1) . 'wincmd w'
    endfor

    " Return to original window
    call win_gotoid(win_getid(l:current_win))
endfunction

]])

vim.fn.RotateWindowLayout()
print("Rotated window layout")
end

return windows
end

```

## Usage Mappings for New Functions:

```

-- Debugging helpers
local debug = debugging_helpers()
vim.keymap.set('n', '<leader>dd', debug.toggle_debug_statements)
vim.keymap.set('v', '<leader>dw', debug.visual_debug_wrapper)

-- Documentation helpers
local docs = documentation_helpers()
vim.keymap.set('n', '<leader>doc', docs.generate_function_docs)
vim.keymap.set('n', '<leader>todo', docs.create_todo_comment)

-- Code analysis
local analysis = code_analysis_helpers()

```

```
vim.keymap.set('n', '<leader>ca', analysis.analyze_function_complexity)
vim.keymap.set('n', '<leader>cs', analysis.find_code_smells)

-- Snippet management
local snippets = snippet_management()
vim.keymap.set('n', '<leader>se', snippets.expand_custom_snippet)
vim.keymap.set('v', '<leader>ss', snippets.save_selection_as_snippet)

-- Advanced window management
local windows = advanced_window_management()
vim.keymap.set('n', '<leader>wb', windows.balance_windows_smart)
vim.keymap.set('n', '<leader>ws', windows.create_scratch_buffer)
vim.keymap.set('n', '<leader>wr', windows.rotate_windows)
```

These additional functions cover advanced debugging, documentation generation, code analysis, snippet management, and sophisticated window operations - providing a comprehensive toolkit for professional development workflows!

---