

with-deepseek

in state of neovim lua configuration, suggest me several custom functions that are plug-and-play

awesome, try to discover more useful yet ergonomic functions

based on both previous responses, suggest me functions that are still not mentioned yet

what about more functions, but inspired by some plugins' functionality

Here are several plug-and-play custom functions for Neovim Lua configuration that you can drop right into your config:

1. Enhanced Buffer Management

```
-- Toggle between current and last buffer
function _G.toggle_last_buffer()
  local current_buf = vim.api.nvim_get_current_buf()
  local last_buf = vim.fn.bufnr('#')

  if last_buf ~= -1 and last_buf ~= current_buf and vim.fn.buflisted(last_buf) == 1
  then
    vim.cmd('buffer #')
  else
    vim.notify("No alternate buffer found", vim.log.levels.WARN)
  end
end

-- Close buffer without closing window
function _G.close_buffer_keep_window()
  local buftype = vim.bo.buftype
  local modified = vim.bo.modified

  if modified then
    vim.ui.input({
      prompt = "Buffer has unsaved changes. Save? (y/n/c): ",
    }, function(input)
      if input == "y" then
        vim.cmd('w | bd')
      elseif input == "n" then
        vim.cmd('bd!')
      end
    end)
  elseif buftype == "terminal" then
    vim.cmd('bd!')
  else
    vim.cmd('bd')
  end
end
```

2. Quick Fix & Location List Utilities

```
-- Toggle quickfix list
function _G.toggle_quickfix()
    local qf_exists = false
    for _, win in pairs(vim.fn.getwininfo()) do
        if win["quickfix"] == 1 then
            qf_exists = true
            break
        end
    end
    if qf_exists then
        vim.cmd('cclose')
        return
    end
    if not vim.tbl_isempty(vim.fn.getqflist()) then
        vim.cmd('copen')
    end
end

-- Toggle location list for current window
function _G.toggle_location_list()
    local loc_exists = false
    for _, win in pairs(vim.fn.getwininfo()) do
        if win["loclist"] == 1 then
            loc_exists = true
            break
        end
    end
    if loc_exists then
        vim.cmd('lclose')
        return
    end
    local loc_list = vim.fn.getloclist(0)
    if not vim.tbl_isempty(loc_list) then
        vim.cmd('lopen')
    end
end
```

3. File Operations

```
-- Create directory of current file if it doesn't exist
function _G.create_current_file_dir()
    local file_path = vim.fn.expand('%:p')
    local dir_path = vim.fn.fnamemodify(file_path, ':h')

    if file_path == "" then
        vim.notify("No file name", vim.log.levels.ERROR)
        return
    end
end
```

```

end

if vim.fn.isdirectory(dir_path) == 0 then
    vim.fn.mkdir(dir_path, 'p')
    vim.notify("Created directory: " .. dir_path)
end
end

-- Toggle between header and source files (C/C++)
function _G.toggle_header_source()
    local current_file = vim.fn.expand('%:t')
    local extension = vim.fn.expand('%:e')
    local base_name = vim.fn.expand('%:t:r')

    local alternatives = {
        h = { 'c', 'cpp', 'cc', 'cxx' },
        c = { 'h' },
        cpp = { 'h', 'hpp' },
        cc = { 'h' },
        cxx = { 'h' },
        hpp = { 'cpp', 'cc', 'cxx' }
    }

    local alt_extensions = alternatives[extension]
    if not alt_extensions then
        vim.notify("No alternative file type found", vim.log.levels.WARN)
        return
    end

    for _, ext in ipairs(alt_extensions) do
        local alt_file = base_name .. '.' .. ext
        if vim.fn.filereadable(alt_file) == 1 then
            vim.cmd('edit ' .. alt_file)
            return
        end
    end

    -- If no existing file found, create the most common alternative
    vim.ui.input({
        prompt = "Alternative file doesn't exist. Create " .. alt_extensions[1] .. "?
(y/n): ",
    }, function(input)
        if input == "y" then
            vim.cmd('edit ' .. base_name .. '.' .. alt_extensions[1])
        end
    end)
end

```

4. Window Management

```

-- Resize windows to equal sizes
function _G.equalize_windows()
    vim.cmd('wincmd =')
end

-- Easy window movement between splits
function _G.move_window(direction)
    local current_win = vim.api.nvim_get_current_win()
    vim.cmd('wincmd ' .. direction)
    local new_win = vim.api.nvim_get_current_win()

    if current_win == new_win then
        vim.notify("No window in that direction", vim.log.levels.WARN)
    end
end

```

5. Text Manipulation

```

-- Toggle between true/false, yes/no, on/off, etc.
function _G.toggle_boolean()
    local replacements = {
        ['true'] = 'false',
        ['false'] = 'true',
        ['yes'] = 'no',
        ['no'] = 'yes',
        ['on'] = 'off',
        ['off'] = 'on',
        ['enable'] = 'disable',
        ['disable'] = 'enable',
        ['0'] = '1',
        ['1'] = '0'
    }

    local line = vim.api.nvim_get_current_line()
    local col = vim.api.nvim_win_get_cursor(0)[2] + 1
    local word = vim.fn.expand('<cword>')

    if replacements[word:lower()] then
        local new_word = replacements[word:lower()]
        -- Preserve case
        if word:sub(1, 1):match('%u') then
            new_word = new_word:sub(1, 1):upper() .. new_word:sub(2)
        end

        local start_col = vim.fn.match(line:sub(1, col), '\\k*$')
        local end_col = start_col + #word

        local new_line = line:sub(1, start_col) .. new_word .. line:sub(end_col + 1)
        vim.api.nvim_set_current_line(new_line)
    end
end

```

```

else
    vim.notify("No boolean word under cursor", vim.log.levels.WARN)
end
end

-- Duplicate current line and comment the original
function _G.duplicate_and_comment()
    local filetype = vim.bo.filetype
    local commentstring = vim.bo.commentstring or '%%s'

    local current_line = vim.api.nvim_get_current_line()
    vim.api.nvim_set_current_line(string.format(commentstring, current_line))
    vim.cmd('normal! o' .. current_line)
end

```

6. Visual Mode Utilities

```

-- Search for selected text in visual mode
function _G.visual_selection_search()
    local saved_register = vim.fn.getreg('v')
    vim.cmd('normal! "vy')
    local selected = vim.fn.getreg('v')
    vim.fn.setreg('v', saved_register)

    if #selected > 0 then
        vim.cmd('/') .. vim.fn.escape(selected, '\\')
    end
end

-- Sort selected lines in visual mode
function _G.sort_visual_lines()
    vim.cmd("<<,>>sort")
end

```

7. Keybinding Setup Helper

```

-- Easy keymap function with options
function _G.map(mode, lhs, rhs, opts)
    local options = { noremap = true, silent = true }
    if opts then
        options = vim.tbl_extend('force', options, opts)
    end
    vim.keymap.set(mode, lhs, rhs, options)
end

-- Example usage:
-- map('n', '<leader>bb', toggle_last_buffer)

```

```
-- map('n', '<leader>bd', close_buffer_keep_window)
-- map('n', '<leader>q', toggle_quickfix)
```

Usage Examples:

Add these to your `init.lua` and set up keybindings:

```
-- Buffer management
vim.keymap.set('n', '<leader>bb', '<cmd>lua toggle_last_buffer(<cr>')
vim.keymap.set('n', '<leader>bd', '<cmd>lua close_buffer_keep_window(<cr>')

-- Quickfix
vim.keymap.set('n', '<leader>q', '<cmd>lua toggle_quickfix(<cr>')

-- File operations
vim.keymap.set('n', '<leader>md', '<cmd>lua create_current_file_dir(<cr>')
vim.keymap.set('n', '<leader>ah', '<cmd>lua toggle_header_source(<cr>')

-- Text manipulation
vim.keymap.set('n', '<leader>tb', '<cmd>lua toggle_boolean(<cr>')
```

These functions are self-contained and should work immediately in any Neovim Lua configuration!

Absolutely! Here are more advanced yet ergonomic functions that enhance daily Neovim workflow:

1. Smart Cursor Position Memory

```
-- Remember cursor position when switching buffers and restore it
local cursor_positions = {}

function _G.save_cursor_position()
  local buf = vim.api.nvim_get_current_buf()
  local pos = vim.api.nvim_win_get_cursor(0)
  cursor_positions[buf] = pos
end

function _G.restore_cursor_position()
  local buf = vim.api.nvim_get_current_buf()
  if cursor_positions[buf] then
    vim.api.nvim_win_set_cursor(0, cursor_positions[buf])
  end
end

-- Auto commands to make it work automatically
vim.cmd([[
  augroup cursor_memory
    autocmd!
    autocmd BufLeave * lua save_cursor_position()
    autocmd BufEnter * lua restore_cursor_position()
  endaugroup
]])
```

```
augroup END
]])
```

2. Intelligent Paste System

```
-- Smart paste that handles indentation properly
function _G.smart_paste()
    local mode = vim.fn.mode()
    local register = vim.fn.getreg('')
    local register_type = vim.fn.getregtype('')

    if mode == "v" or mode == "V" then
        -- In visual mode, replace selection
        vim.cmd('normal! "_d')
    end

    if register_type == "V" then -- Linewise paste
        local current_line = vim.fn.line('.')
        local current_indent = vim.fn.indent(current_line)

        vim.cmd('normal! ""p')

        -- Adjust indentation for pasted lines
        local last_line = vim.fn.line('.')
        for i = current_line + 1, last_line do
            local line_indent = vim.fn.indent(i)
            if line_indent > 0 then
                local new_indent = math.max(0, line_indent - (vim.fn.indent(current_line) -
current_indent))
                vim.fn.setline(i, string.rep(' ', new_indent) ..
vim.fn.getline(i):match('^%s*(.*)'))
            end
        end
    else
        vim.cmd('normal! ""p')
    end
end

-- Paste and immediately fix indentation
function _G.paste_and_indent()
    vim.cmd('normal! ""p')
    vim.cmd('normal! `[v`=')
end
```

3. Context-Aware Code Navigation

```
-- Jump to definition or declaration based on context
function _G.smart_jump()
```

```

local filetype = vim.bo.filetype
local word_under_cursor = vim.fn.expand('<cword>')

-- Try LSP definition first
local clients = vim.lsp.get_active_clients()
if #clients > 0 then
    vim.lsp.buf.definition()
    return
end

-- Fallback to tags
if vim.fn.filereadable('tags') == 1 or vim.fn.filereadable('./tags') == 1 then
    vim.cmd('tag ' .. word_under_cursor)
else
    -- Ultimate fallback - search for word
    vim.cmd('normal! *')
end
end

-- Return from jump with context
function _G.smart_return()
    local jump_list = vim.fn.getjumplist()
    if #jump_list[1] > 1 then
        vim.cmd('normal! <c-o>')
    else
        vim.notify("No previous jump location", vim.log.levels.INFO)
    end
end
end

```

4. Dynamic Terminal Management

```

-- Toggle terminal at the bottom with smart sizing
function _G.toggle_terminal()
    local term_buf = nil
    local term_win = nil

    -- Find existing terminal window
    for _, win in ipairs(vim.api.nvim_list_wins()) do
        local buf = vim.api.nvim_win_get_buf(win)
        local buftype = vim.api.nvim_buf_get_option(buf, 'buftype')
        if buftype == 'terminal' then
            term_buf = buf
            term_win = win
            break
        end
    end

    if term_win and vim.api.nvim_win_is_valid(term_win) then
        -- Toggle terminal window
        if vim.api.nvim_get_current_win() == term_win then

```



```

        vim.api.nvim_win_hide(term_win)
    else
        vim.api.nvim_set_current_win(term_win)
    end
else
    -- Create new terminal
    vim.cmd('botright split | terminal')
    term_buf = vim.api.nvim_get_current_buf()
    term_win = vim.api.nvim_get_current_win()

    -- Set terminal height based on window size
    local win_height = vim.api.nvim_win_get_height(0)
    local term_height = math.floor(win_height * 0.3)
    vim.cmd('resize ' .. term_height)

    -- Enter terminal mode
    vim.cmd('startinsert')
end
end

-- Send visual selection to terminal
function _G.send_to_terminal()
    local saved_register = vim.fn.getreg('')
    vim.cmd('normal! gv"ty')
    local selected_text = vim.fn.getreg('t')
    vim.fn.setreg('', saved_register)

    -- Find terminal buffer
    for _, buf in ipairs(vim.api.nvim_list_bufs()) do
        if vim.api.nvim_buf_is_valid(buf) and vim.api.nvim_buf_get_option(buf,
'buftype') == 'terminal' then
            local chan = vim.api.nvim_buf_get_option(buf, 'channel')
            if chan and chan > 0 then
                vim.api.nvim_chan_send(chan, selected_text .. '\r')
                vim.notify("Sent to terminal", vim.log.levels.INFO)
                return
            end
        end
    end
end

vim.notify("No active terminal found", vim.log.levels.ERROR)
end

```

5. Intelligent Search & Replace

```

-- Search and replace with preview
function _G.preview_replace()
    local word_under_cursor = vim.fn.expand('<cword>')

    vim.ui.input({

```

```

    prompt = 'Replace "' .. word_under_cursor .. '" with: ',
    default = '',
}, function(replacement)
    if replacement and replacement ~= '' then
        -- Show occurrences
        vim.cmd('match Search /' .. vim.fn.escape(word_under_cursor, '/') .. '/')

        vim.ui.input({
            prompt = 'Replace all? (y/n): ',
        }, function(confirm)
            vim.cmd('match none')
            if confirm == 'y' then
                vim.cmd('%s/' .. vim.fn.escape(word_under_cursor, '/') .. '/' ..
replacement .. '/g')
                vim.notify("Replaced all occurrences")
            end
        end)
    end
end)
end

-- Incremental search counter
function _G.show_search_count()
    local search_term = vim.fn.getreg('/')
    if search_term == '' then return end

    local line1, line2 = 1, vim.fn.line('$')
    local count = 0
    local current_line = vim.fn.line('.')

    for i = line1, line2 do
        local line = vim.fn.getline(i)
        local occurrences = select(2, string.gsub(line, search_term, ''))
        count = count + occurrences
    end

    vim.notify(string.format("%s: %d matches in buffer", search_term, count))
end

```

6. Window Management 2.0

```

-- Rotate windows clockwise
function _G.rotate_windows()
    local current_win = vim.api.nvim_get_current_win()
    local windows = vim.api.nvim_list_wins()
    local layout = {}

    -- Get window positions
    for _, win in ipairs(windows) do
        local config = vim.api.nvim_win_get_config(win)
    end

```

```

    if config.relative == '' then -- Only non-floating windows
        table.insert(layout, {win = win, row = config.row, col = config.col})
    end
end

-- Sort by position and rotate buffers
table.sort(layout, function(a, b)
    if a.row == b.row then
        return a.col < b.col
    end
    return a.row < b.row
end)

if #layout > 1 then
    local buffers = {}
    for i, item in ipairs(layout) do
        buffers[i] = vim.api.nvim_win_get_buf(item.win)
    end

    -- Rotate buffers
    local last_buf = buffers[#buffers]
    for i = #buffers, 2, -1 do
        vim.api.nvim_win_set_buf(layout[i].win, buffers[i-1])
    end
    vim.api.nvim_win_set_buf(layout[1].win, last_buf)
end
end

-- Maximize window toggle
function _G.toggle_maximize()
    if _G.maximized_win then
        vim.api.nvim_win_set_config(_G.maximized_win, _G.original_config)
        _G.maximized_win = nil
        _G.original_config = nil
    else
        _G.maximized_win = vim.api.nvim_get_current_win()
        _G.original_config = vim.api.nvim_win_get_config(_G.maximized_win)

        local width = vim.o.columns
        local height = vim.o.lines - vim.o.cmdheight
        vim.api.nvim_win_set_config(_G.maximized_win, {
            style = 'minimal',
            width = width,
            height = height,
            row = 0,
            col = 0,
            relative = 'editor'
        })
    end
end
end

```

7. Code Diagnostics Helper

```
-- Navigate through diagnostics with preview
function _G.diagnostic_navigation(direction)
    local severity_levels = { "Error", "Warn", "Info", "Hint" }
    local diagnostics = vim.diagnostic.get(0, { severity =
vim.diagnostic.severity.ERROR })

    if #diagnostics == 0 then
        diagnostics = vim.diagnostic.get(0, { severity = vim.diagnostic.severity.WARN })
    end

    if #diagnostics == 0 then
        diagnostics = vim.diagnostic.get(0, { severity = vim.diagnostic.severity.INFO })
    end

    if #diagnostics == 0 then
        diagnostics = vim.diagnostic.get(0, { severity = vim.diagnostic.severity.HINT })
    end

    if #diagnostics > 0 then
        if direction == "next" then
            vim.diagnostic.goto_next()
        else
            vim.diagnostic.goto_prev()
        end
    end

    -- Show diagnostic in floating window
    local bufnr = vim.api.nvim_get_current_buf()
    local opts = {
        focusable = false,
        close_events = { "BufLeave", "CursorMoved", "InsertEnter", "FocusLost" },
        border = 'rounded',
        source = 'always',
        prefix = ' ',
        scope = 'cursor',
    }
    vim.diagnostic.open_float(opts)
else
    vim.notify("No diagnostics found", vim.log.levels.INFO)
end
end
```

8. Session Management

```
-- Quick session save/load
function _G.quick_session(session_name)
    local session_dir = vim.fn.stdpath('data') .. '/sessions'
    if vim.fn.isdirectory(session_dir) == 0 then
```

```

    vim.fn.mkdir(session_dir, 'p')
end

if not session_name then
    session_name = vim.fn.getcwd():gsub('[^%w_]', '_')
end

local session_file = session_dir .. '/' .. session_name .. '.vim'

if vim.fn.filereadable(session_file) == 1 then
    -- Load session
    vim.cmd('source ' .. session_file)
    vim.notify("Loaded session: " .. session_name)
else
    -- Save session
    vim.cmd('mksession! ' .. session_file)
    vim.notify("Saved session: " .. session_name)
end
end

```

Keybinding Suggestions:

```

-- Smart navigation
vim.keymap.set('n', 'gd', '<cmd>lua smart_jump(<cr>')
vim.keymap.set('n', 'gD', '<cmd>lua vim.lsp.buf.declaration(<cr>')
vim.keymap.set('n', '<C-t>', '<cmd>lua smart_return(<cr>')

-- Terminal
vim.keymap.set('n', '<leader>t', '<cmd>lua toggle_terminal(<cr>')
vim.keymap.set('v', '<leader>st', '<cmd>lua send_to_terminal(<cr>')

-- Search & replace
vim.keymap.set('n', '<leader>r', '<cmd>lua preview_replace(<cr>')
vim.keymap.set('n', '<leader>sc', '<cmd>lua show_search_count(<cr>')

-- Window management
vim.keymap.set('n', '<leader>wr', '<cmd>lua rotate_windows(<cr>')
vim.keymap.set('n', '<leader>wm', '<cmd>lua toggle_maximize(<cr>')

-- Diagnostics
vim.keymap.set('n', ']d', '<cmd>lua diagnostic_navigation("next")<cr>')
vim.keymap.set('n', '[d', '<cmd>lua diagnostic_navigation("prev")<cr>')

-- Sessions
vim.keymap.set('n', '<leader>ss', '<cmd>lua quick_session(<cr>')

```

These functions provide intelligent, context-aware behavior that adapts to your workflow while remaining completely plug-and-play!

Here are more unique and ergonomic functions that haven't been mentioned yet:

1. Intelligent Code Annotation

```
-- Auto-generate TODO comments with context
function _G.smart_todo()
    local filetype = vim.bo.filetype
    local comment_chars = {
        lua = '--',
        python = '#',
        javascript = '///',
        typescript = '///',
        c = '///',
        cpp = '///',
        java = '///',
        go = '///',
        rust = '///',
        vim = '',
    }

    local comment = comment_chars[filetype] or '///'
    local todo_types = {'TODO', 'FIXME', 'NOTE', 'HACK', 'BUG', 'OPTIMIZE'}

    vim.ui.select(todo_types, {
        prompt = 'Select TODO type:',
    }, function(choice)
        if choice then
            vim.ui.input({
                prompt = choice .. ': ',
            }, function(input)
                if input then
                    local line = comment .. ' ' .. choice .. ': ' .. input
                    vim.api.nvim_put({line}, 'l', true, true)
                end
            end)
        end
    end)
end

-- Annotate current line with timestamp
function _G.annotate_with_timestamp()
    local comment_chars = {
        lua = '--',
        python = '#',
        javascript = '///',
        typescript = '///',
        c = '///',
        cpp = '///',
        vim = '',
        default = '#'
    }
end
```

```

local ft = vim.bo.filetype
local comment = comment_chars[ft] or comment_chars.default
local timestamp = os.date('%Y-%m-%d %H:%M:%S')
local current_line = vim.api.nvim_get_current_line()

if current_line:match('^%s*$') then
    vim.api.nvim_set_current_line(comment .. ' ' .. timestamp)
else
    vim.api.nvim_set_current_line(current_line .. ' ' .. comment .. ' ' ..
timestamp)
end
end

```

2. Dynamic Workspace Management

```

-- Switch between project roots automatically
function _G.detect_project_root()
    local root_patterns = {
        '.git', 'package.json', 'Cargo.toml', 'pyproject.toml',
        'requirements.txt', 'Makefile', 'CMakeLists.txt', 'go.mod'
    }

    local current_file = vim.fn.expand('%:p')
    if current_file == '' then
        current_file = vim.fn.getcwd()
    end

    for _, pattern in ipairs(root_patterns) do
        local root = vim.fn.finddir(pattern, current_file .. ';')
        if root ~= '' then
            return vim.fn.fnamemodify(root, ':h')
        end
    end

    return vim.fn.getcwd()
end

function _G.switch_project_root()
    local new_root = detect_project_root()
    if new_root ~= vim.fn.getcwd() then
        vim.cmd('cd ' .. vim.fn.fnameescape(new_root))
        vim.notify('Switched to project: ' .. vim.fn.fnamemodify(new_root, ':t'))
    end
end

```

3. Visual Feedback System

```

-- Highlight current line number temporarily
function _G.flash_line_feedback()
    local current_win = vim.api.nvim_get_current_win()
    local ns = vim.api.nvim_create_namespace('flash_line')

    -- Highlight current line number
    vim.api.nvim_win_set_hl_ns(current_win, ns)
    vim.api.nvim_buf_set_extmark(0, ns, vim.fn.line('.') - 1, 0, {
        number_hl_group = 'Visual',
        line_hl_group = 'Visual',
        end_line = vim.fn.line('.'),
        priority = 1000,
    })

    -- Clear after delay
    vim.defer_fn(function()
        vim.api.nvim_buf_clear_namespace(0, ns, 0, -1)
    end, 300)
end

-- Visual feedback for yank operations
function _G.visual_yank_feedback()
    local saved_eventignore = vim.o.eventignore
    vim.o.eventignore = 'all'

    -- Flash the yanked area
    vim.cmd('normal! `[v`]')
    vim.cmd('redraw')
    vim.fn.sleep(100)
    vim.cmd('normal! <Esc>')

    vim.o.eventignore = saved_eventignore
    vim.notify('Yanked ' .. vim.v.count1 .. ' lines')
end

```

4. Smart Text Objects

```

-- Select increasingly larger context (word -> paragraph -> function)
function _G.expand_selection()
    local mode = vim.fn.mode()

    if mode == 'v' or mode == 'V' then
        -- Already in visual mode, expand selection
        local current_pos = vim.fn.getpos('.')
        local start_pos = vim.fn.getpos("'<")
        local end_pos = vim.fn.getpos("'>")

        -- Try to expand to different text objects
        vim.cmd('normal! o')
    end
end

```



```

local new_end_pos = vim.fn.getpos("'.")

if new_end_pos[2] == end_pos[2] and new_end_pos[3] == end_pos[3] then
    -- No expansion happened, try larger units
    vim.fn.setpos('.', current_pos)
    vim.cmd('normal! vip')
end
else
    -- Start visual mode and select word
    vim.cmd('normal! viw')
end
end

-- Select between matching pairs
function _G.select_between_pairs()
    local pairs = {'(' = ')', '[' = ']', '{' = '}', '<' = '>', '"' = '"',
['''' = ''', ['``' = ``}]
    local line = vim.api.nvim_get_current_line()
    local col = vim.fn.col('.')
    local char = line:sub(col, col)

    if pairs[char] then
        vim.cmd('normal! va' .. char)
    else
        -- Find the opening pair
        local stack = {}
        for i = 1, #line do
            local c = line:sub(i, i)
            if pairs[c] then
                table.insert(stack, c)
            elseif not next(stack) then
                -- Skip if stack is empty
            else
                local last = stack[#stack]
                if c == pairs[last] then
                    table.remove(stack)
                    if #stack == 0 then
                        -- Found matching pair, select between them
                        vim.fn.cursor(vim.fn.line('.'), i)
                        vim.cmd('normal! v')
                        vim.fn.cursor(vim.fn.line('.'), col)
                        return
                    end
                end
            end
        end
        vim.notify('No matching pair found', vim.log.levels.WARN)
    end
end
end

```

5. Context-Aware Code Formatting

```
-- Format selection or buffer based on context
function _G.smart_format()
    local mode = vim.fn.mode()

    if mode == 'v' or mode == 'V' then
        -- Format visual selection
        local start_line = vim.fn.line("'<")
        local end_line = vim.fn.line("'>")

        -- Try LSP range formatting first
        local clients = vim.lsp.get_active_clients()
        if #clients > 0 then
            vim.lsp.buf.format({ range = {
                start = { start_line - 1, 0 },
                ['end'] = { end_line - 1, 0 }
            }})
            return
        end

        -- Fallback to external formatters
        vim.cmd(start_line .. ',' .. end_line .. '!')
    else
        -- Format entire buffer
        local clients = vim.lsp.get_active_clients()
        if #clients > 0 then
            vim.lsp.buf.format()
        else
            vim.cmd('normal! gg=G')
        end
    end
end

-- Toggle between compact and expanded formatting
function _G.toggle_code_density()
    if not _G.code_dense then
        _G.code_dense = false
    end

    if _G.code_dense then
        -- Expand code (add more spacing)
        vim.cmd([[%s/}\r{/g]])
        vim.cmd([[%s/;/\r/g]])
        vim.notify('Code expanded')
    else
        -- Compact code (remove extra spacing)
        vim.cmd([[%s/\s*{/g]])
        vim.cmd([[%s/;\n\s*/; /g]])
        vim.notify('Code compacted')
    end
end
```

```

end

_G.code_dense = not _G.code_dense
end

```

6. Intelligent File Creation

```

-- Create file with template based on extension
function _G.create_file_with_template()
    vim.ui.input({
        prompt = 'File name: ',
    }, function(filename)
        if filename then
            local extension = filename:match('%.(%w+)$') or ''
            local templates = {
                py = '#!/usr/bin/env python3\n# -*- coding: utf-8 -*-\n\n',
                lua = '-- ' .. filename .. '\n\n',
                sh = '#!/bin/bash\n\n',
                js = '// ' .. filename .. '\n\n',
                html = '<!DOCTYPE html>\n<html>\n<head>\n
<title>Document</title>\n</head>\n<body>\n    \n</body>\n</html>',
            }

            vim.cmd('edit ' .. filename)
            local template = templates[extension]
            if template then
                vim.api.nvim_buf_set_lines(0, 0, -1, false, vim.split(template, '\n'))
                vim.cmd('write')
            end
        end
    end)
end

-- Create related test file
function _G.create_test_file()
    local current_file = vim.fn.expand('%:t')
    local base_name = vim.fn.expand('%:t:r')
    local extension = vim.fn.expand('%:e')

    local test_patterns = {
        lua = 'spec/' .. base_name .. '_spec.' .. extension,
        py = 'test_' .. base_name .. '.' .. extension,
        js = '__tests__/' .. base_name .. '.test.' .. extension,
        ts = '__tests__/' .. base_name .. '.test.' .. extension,
    }

    local test_file = test_patterns[extension] or 'test_' .. current_file

    vim.ui.input({
        prompt = 'Test file: ',
    }, function(filename)
        if filename then
            vim.cmd('edit ' .. filename)
            vim.cmd('write')
        end
    end)
end

```

```

    default = test_file,
}, function(input)
    if input then
        vim.cmd('edit ' .. input)
        vim.notify('Created test file: ' .. input)
    end
end)
end
end

```

7. Advanced Search & Navigation

```

-- Search for visually selected text across workspace
function _G.visual_search_workspace()
    local saved_register = vim.fn.getreg('')
    vim.cmd('normal! "vy')
    local selected = vim.fn.getreg('')
    vim.fn.setreg('', saved_register)

    if #selected > 0 then
        -- Use grep or similar tool
        local cmd = 'rg --vimgrep "' .. vim.fn.escape(selected, '"') .. '"'
        vim.cmd('cgetexpr system("'" .. cmd .. "'')')
        if not vim.tbl_isempty(vim.fn.getqflist()) then
            vim.cmd('copen')
        else
            vim.notify('No matches found', vim.log.levels.WARN)
        end
    end
end

-- Navigate to related files (config, docs, etc.)
function _G.find_related_files()
    local current_file = vim.fn.expand('%:t')
    local base_name = vim.fn.expand('%:t:r')
    local patterns = {
        'README*',
        'CHANGELOG*',
        'CONTRIBUTING*',
        'LICENSE*',
        'Makefile',
        'package.json',
        'pyproject.toml',
        'requirements.txt',
        '*.md',
        'docs/**/*.md',
    }

    local matches = {}
    for _, pattern in ipairs(patterns) do
        local files = vim.fn.glob(pattern, true, true)
    end
end

```

```

    for _, file in ipairs(files) do
        table.insert(matches, file)
    end
end

if #matches > 0 then
    vim.ui.select(matches, {
        prompt = 'Related files:',
    }, function(choice)
        if choice then
            vim.cmd('edit ' .. choice)
        end
    end)
else
    vim.notify('No related files found', vim.log.levels.INFO)
end
end

```

8. Smart Macro System

```

-- Record macro with automatic naming
function _G.smart_macro_record()
    local macros = vim.fn.getreg('a') ~= '' and 'a' or
        vim.fn.getreg('b') ~= '' and 'b' or
        vim.fn.getreg('c') ~= '' and 'c' or 'a'

    vim.notify('Recording macro in register ' .. macros)
    vim.cmd('normal! q' .. macros)
end

-- Play macro with visual feedback
function _G.smart_macro_play()
    local register = vim.fn.reg_recording()
    if register == '' then
        register = 'a'
    end

    if vim.fn.getreg(register) == '' then
        vim.notify('No macro recorded in register ' .. register, vim.log.levels.WARN)
        return
    end

    vim.notify('Playing macro ' .. register)
    vim.cmd('normal! @' .. register)
end

-- Apply macro to all matches of last search
function _G.macro_to_search_matches()
    local register = vim.fn.reg_recording()
    if register == '' then

```

```

    register = 'a'
end

if vim.fn.getreg(register) == '' then
    vim.notify('No macro recorded', vim.log.levels.WARN)
    return
end

local search_term = vim.fn.getreg('/')
if search_term == '' then
    vim.notify('No search pattern', vim.log.levels.WARN)
    return
end

vim.cmd('normal! gg')
vim.cmd('normal! /' .. search_term .. '\r')

while true do
    vim.cmd('normal! @' .. register)
    local result = vim.fn.search(search_term, 'W')
    if result == 0 then break end
end

vim.notify('Applied macro to all matches')
end

```

9. Buffer Context Preservation

```

-- Save buffer context (folds, cursor, etc.) when switching
local buffer_context = {}

function _G.save_buffer_context()
    local buf = vim.api.nvim_get_current_buf()
    buffer_context[buf] = {
        cursor = vim.api.nvim_win_get_cursor(0),
        folds = vim.fn.getwininfo()[1].foldmethod ~= '' and vim.cmd('mkview') or nil,
        scroll = vim.fn.line('w0')
    }
end

function _G.restore_buffer_context()
    local buf = vim.api.nvim_get_current_buf()
    local context = buffer_context[buf]
    if context then
        if context.folds then
            vim.cmd('loadview')
        end
        vim.api.nvim_win_set_cursor(0, context.cursor)
    end
end

```

```
-- Auto commands for context preservation
vim.cmd([[
  augroup buffer_context
    autocmd!
    autocmd BufLeave * lua save_buffer_context()
    autocmd BufEnter * lua restore_buffer_context()
  augroup END
]])
```

Keybinding Suggestions:

```
-- Smart annotations
vim.keymap.set('n', '<leader>ta', '<cmd>lua smart_todo()<cr>')
vim.keymap.set('n', '<leader>tt', '<cmd>lua annotate_with_timestamp()<cr>')

-- Workspace management
vim.keymap.set('n', '<leader>pw', '<cmd>lua switch_project_root()<cr>')

-- Visual feedback
vim.keymap.set('n', '<leader>y', '<cmd>lua visual_yank_feedback()<cr>', {silent = true})

-- Text objects
vim.keymap.set('n', 'v', '<cmd>lua expand_selection()<cr>', {silent = true})
vim.keymap.set('n', 'va', '<cmd>lua select_between_pairs()<cr>', {silent = true})

-- Smart formatting
vim.keymap.set({'n', 'v'}, '<leader>f', '<cmd>lua smart_format()<cr>')
vim.keymap.set('n', '<leader>fd', '<cmd>lua toggle_code_density()<cr>')

-- File creation
vim.keymap.set('n', '<leader>nf', '<cmd>lua create_file_with_template()<cr>')
vim.keymap.set('n', '<leader>nt', '<cmd>lua create_test_file()<cr>')

-- Advanced search
vim.keymap.set('n', '<leader>sw', '<cmd>lua visual_search_workspace()<cr>')
vim.keymap.set('n', '<leader>fr', '<cmd>lua find_related_files()<cr>')

-- Smart macros
vim.keymap.set('n', '<leader>qr', '<cmd>lua smart_macro_record()<cr>')
vim.keymap.set('n', '<leader>qp', '<cmd>lua smart_macro_play()<cr>')
vim.keymap.set('n', '<leader>qa', '<cmd>lua macro_to_search_matches()<cr>')
```

These functions provide sophisticated, context-aware behaviors that adapt intelligently to your workflow while maintaining the plug-and-play nature!

Here are plugin-inspired functions that replicate popular plugin functionality in pure Lua:

1. Telescope-inspired Fuzzy Finder

```
-- Simple file fuzzy finder
function _G.fuzzy_find_files()
    local scan = require('plenary.scandir')
    local Path = require('plenary.path')

    local cwd = vim.fn.getcwd()
    local files = scan.scan_dir(cwd, {
        hidden = true,
        add_dirs = true,
        depth = 3,
    })

    -- Filter and format files
    local formatted_files = {}
    for _, file in ipairs(files) do
        local relative_path = Path:new(file):make_relative(cwd)
        table.insert(formatted_files, relative_path)
    end

    vim.ui.select(formatted_files, {
        prompt = 'Find File:',
        format_item = function(item)
            return ' ' .. item
        end,
    }, function(choice)
        if choice then
            vim.cmd('edit ' .. choice)
        end
    end)
end

-- Live grep alternative
function _G.live_grep()
    vim.ui.input({
        prompt = 'Grep Pattern: ',
    }, function(pattern)
        if pattern then
            local Job = require('plenary.job')
            local results = {}

            Job:new({
                command = 'rg',
                args = {'--vimgrep', pattern},
                cwd = vim.fn.getcwd(),
                on_stdout = function(_, data)
                    if data then table.insert(results, data) end
                end,
            }):sync()
        end
    end)
end
```



```

if #results > 0 then
    vim.ui.select(results, {
        prompt = 'Grep Results:',
        format_item = function(item)
            local filename, lnum, col, text = item:match('([^\:]+):(%d+):(%d+):(.+)')
            return string.format('%s:%s:%s  %s', filename, lnum, col, text)
        end,
    }, function(choice)
        if choice then
            local filename, lnum = choice:match('([^\:]+):(%d+):')
            vim.cmd('edit ' .. filename)
            vim.fn.cursor(tonumber(lnum), 1)
        end
    end)
else
    vim.notify('No matches found', vim.log.levels.WARN)
end
end
end)
end

```

2. WhichKey-inspired Keybinding Helper

```

-- Show available keybindings with descriptions
function _G.show_keybindings()
    local keymaps = vim.api.nvim_get_keymap('n')
    local grouped = {}

    for _, map in ipairs(keymaps) do
        local leader_match = map.lhs:match('^<leader>(.)')
        if leader_match then
            local group = leader_match
            if not grouped[group] then grouped[group] = {} end
            table.insert(grouped[group], {
                key = map.lhs,
                desc = map.desc or 'No description',
                rhs = map.rhs
            })
        end
    end

    local choices = {}
    for group, maps in pairs(grouped) do
        table.insert(choices, group .. ' - ' .. #maps .. ' keymaps')
        for _, map in ipairs(maps) do
            table.insert(choices, ' ' .. map.key .. ' → ' .. map.desc)
        end
    end
end

```

```

vim.ui.select(choices, {
  prompt = 'Available Keybindings:',
}, function(choice)
  if choice and choice:match('^<leader>') then
    local key = choice:match('^(%S+)')
    vim.api.nvim_feedkeys(vim.api.nvim_replace_termcodes(key, true, false, true),
'n', false)
  end
end)
end

```

3. NeoTree-inspired File Explorer

```

-- Simple file tree explorer
function _G.toggle_file_explorer()
  local bufname = vim.fn.bufname()
  if bufname:match('File Explorer') then
    vim.cmd('bd')
    return
  end

  -- Create explorer buffer
  vim.cmd('vnew')
  vim.bo.buftype = 'nofile'
  vim.bo.bufhidden = 'wipe'
  vim.bo.swapfile = false
  vim.bo.filetype = 'filetree'

  vim.api.nvim_buf_set_name(0, 'File Explorer')

  local function refresh_tree()
    local lines = {}
    local cwd = vim.fn.getcwd()

    table.insert(lines, '📁 ' .. cwd)
    table.insert(lines, '')

    -- Add directories
    local dirs = vim.fn.globpath(cwd, '*/', 1)
    for dir in vim.gsplit(dirs, '\n', {}) do
      if dir ~= '' then
        local name = vim.fn.fnamemodify(dir, ':t')
        table.insert(lines, '📁 ' .. name)
      end
    end

    -- Add files
    local files = vim.fn.globpath(cwd, '*', 1)
    for file in vim.gsplit(files, '\n', {}) do
      if file ~= '' and vim.fn.isdirectory(file) == 0 then

```

```

        local name = vim.fn.fnamemodify(file, ':t')
        local icon = get_file_icon(name)
        table.insert(lines, ' ' .. icon .. ' ' .. name)
    end
end

vim.api.nvim_buf_set_lines(0, 0, -1, false, lines)

-- Set up keymaps for the explorer
vim.keymap.set('n', '<CR>', function()
    local line = vim.fn.getline('.')
    local name = line:match('[📁📄]%s+(.)$')
    if name then
        if line:match('📁') then
            vim.cmd('cd ' .. name)
            refresh_tree()
        else
            vim.cmd('edit ' .. name)
        end
    end
end, { buffer = true })

vim.keymap.set('n', 'q', '<cmd>bd<cr>', { buffer = true })
end

local function get_file_icon(filename)
    local icons = {
        ['.lua'] = '📄',
        ['.py'] = '📄',
        ['.js'] = '📄',
        ['.ts'] = '📄',
        ['.json'] = '📄',
        ['.md'] = '📄',
        ['.vim'] = '📄',
        ['.gitignore'] = '📄',
    }

    for ext, icon in pairs(icons) do
        if filename:match(ext .. '$') then
            return icon
        end
    end

    return '📄'
end

refresh_tree()
end

```

4. Comment.nvim-inspired Smart Comments

```

-- Toggle comments with context awareness
function _G.toggle_comment()
    local filetype = vim.bo.filetype
    local commentstring = vim.bo.commentstring or '//'
    local left, right = commentstring:match('^(-)%s*(.*)$')

    if not left then
        left = commentstring:gsub('%s', '')
        right = ''
    end

    local mode = vim.fn.mode()

    if mode == 'V' or mode == 'v' then
        -- Visual mode comment toggling
        local start_line = vim.fn.line("'<")
        local end_line = vim.fn.line("'>")

        -- Check if first line is already commented
        local first_line = vim.fn.getline(start_line)
        local is_commented = first_line:match('^%s*' .. vim.pesc(left))

        for i = start_line, end_line do
            local line = vim.fn.getline(i)
            if is_commented then
                -- Uncomment
                local uncommented = line:gsub('^%s*' .. vim.pesc(left), '')
                uncommented = uncommented:gsub(vim.pesc(right) .. '$', '')
                vim.fn.setline(i, uncommented)
            else
                -- Comment
                local indent = line:match('^(%s*)')
                local content = line:match('^%s*(.*)$')
                vim.fn.setline(i, indent .. left .. content .. right)
            end
        end
    else
        -- Normal mode - toggle current line
        local line = vim.fn.getline('.')
        if line:match('^%s*' .. vim.pesc(left)) then
            -- Uncomment
            local uncommented = line:gsub('^%s*' .. vim.pesc(left), '')
            uncommented = uncommented:gsub(vim.pesc(right) .. '$', '')
            vim.fn.setline('.', uncommented)
        else
            -- Comment
            local indent = line:match('^(%s*)')
            local content = line:match('^%s*(.*)$')
            vim.fn.setline('.', indent .. left .. content .. right)
        end
    end
end

```

```
end
end
```

5. AutoPairs-inspired Smart Pairing

```
-- Smart pair completion
function _G.smart_pair(char)
    local pairs = {
        ['('] = ')', ['['] = ']', ['{'] = '}',
        ['"'] = '"', ['"'] = '"', ['`'] = '`',
        ['<'] = '>'
    }

    local line = vim.api.nvim_get_current_line()
    local col = vim.fn.col('.') - 1
    local next_char = line:sub(col + 1, col + 1)

    if pairs[char] and (next_char:match('%s') or next_char == ' ' or next_char ==
pairs[char]) then
        -- Insert both pairs and move cursor between them
        vim.api.nvim_put({char .. pairs[char]}, 'c', false, true)
        vim.cmd('normal! h')
    else
        -- Just insert the character
        vim.api.nvim_put({char}, 'c', false, true)
    end
end

-- Smart pair deletion
function _G.smart_delete()
    local line = vim.api.nvim_get_current_line()
    local col = vim.fn.col('.') - 1
    local current_char = line:sub(col + 1, col + 1)
    local next_char = line:sub(col + 2, col + 2)

    local pairs = {[ '('] = ')', ['['] = ']', ['{'] = '}', ['"'] = '"', ['"'] = '"',
['`'] = '`'}

    if pairs[current_char] and next_char == pairs[current_char] then
        -- Delete both pair characters
        vim.cmd('normal! x')
        vim.cmd('normal! x')
    else
        -- Normal deletion
        vim.cmd('normal! x')
    end
end
```

6. Indent-Blankline-inspired Visual Guides

```

-- Toggle indent guides
function _G.toggle_indent_guides()
    if vim.w.indent_guides then
        vim.cmd('highlight clear IndentGuidesOdd')
        vim.cmd('highlight clear IndentGuidesEven')
        vim.w.indent_guides = false
        vim.notify('Indent guides disabled')
    else
        vim.cmd('highlight IndentGuidesOdd guibg=#1f1f1f ctermbg=234')
        vim.cmd('highlight IndentGuidesEven guibg=#262626 ctermbg=235')
        vim.w.indent_guides = true

        -- Create indent guide matches
        vim.cmd('match IndentGuidesOdd /\%2v\s\{1,\}/')
        vim.cmd('2match IndentGuidesEven /\%4v\s\{1,\}/')
        vim.cmd('3match IndentGuidesOdd /\%6v\s\{1,\}/')

        vim.notify('Indent guides enabled')
    end
end

-- Show current indent level
function _G.show_indent_level()
    local line = vim.api.nvim_get_current_line()
    local indent = line:match('^(%s*)')
    local spaces = #indent
    local tabs = select(2, indent:gsub('\t', ' '))

    if vim.bo.expandtab then
        vim.notify(string.format('Indent: %d spaces', spaces))
    else
        vim.notify(string.format('Indent: %d tabs (%d spaces)', tabs, spaces))
    end
end

```

7. Gitsigns-inspired Git Integration

```

-- Simple git status indicators
function _G.show_git_status()
    local Job = require('plenary.job')

    Job:new({
        command = 'git',
        args = {'status', '--porcelain'},
        cwd = vim.fn.getcwd(),
        on_exit = function(j, return_val)
            if return_val == 0 then
                local changes = {}
                for _, line in ipairs(j:result()) do

```

```

        local status, file = line:match('^(..)%s+(.)$')
        if status and file then
            local icon = get_git_icon(status)
            table.insert(changes, icon .. ' ' .. file .. ' (' .. status .. ')')
        end
    end

    if #changes > 0 then
        vim.ui.select(changes, {
            prompt = 'Git Changes:',
        }, function(choice)
            if choice then
                local file = choice:match('%s+(.)%s+%(')
                vim.cmd('edit ' .. file)
            end
        end)
    else
        vim.notify('No changes', vim.log.levels.INFO)
    end
else
    vim.notify('Not a git repository', vim.log.levels.ERROR)
end
end
}):start()
end

local function get_git_icon(status)
    local icons = {
        [' M'] = '□', -- Modified
        ['M '] = '□',
        [' A'] = '□', -- Added
        ['A '] = '□',
        [' D'] = '□', -- Deleted
        ['D '] = '□',
        [' R'] = '□', -- Renamed
        ['R '] = '□',
        ['??'] = '□', -- Untracked
    }
    return icons[status] or '□'
end

-- Blame current line
function _G.show_git_blame()
    local Job = require('plenary.job')
    local current_file = vim.fn.expand('%:p')
    local current_line = vim.fn.line('.')

    Job:new({
        command = 'git',
        args = {'blame', '--porcelain', '-L', current_line .. ',', .. current_line,
current_file},
        cwd = vim.fn.getcwd(),
    })
end

```

```

on_exit = function(j, return_val)
  if return_val == 0 then
    local result = j:result()
    if #result > 0 then
      local hash = result[1]:match('^(%S+)')
      local author = nil
      local date = nil

      for i = 2, #result do
        if result[i]:match('^author ') then
          author = result[i]:match('^author%s+(.)$')
        elseif result[i]:match('^author-time ') then
          local timestamp = result[i]:match('^author-time%s+(%d+)$')
          if timestamp then
            date = os.date('%Y-%m-%d', tonumber(timestamp))
          end
        end
      end

      if author and date then
        vim.notify(string.format('Blame: %s (%s) %s', author, date, hash:sub(1,
8)))
      end
    end
  end
end
}):start()
end

```

8. LuaLine-inspired Statusline

```

-- Dynamic statusline components
function _G.custom_statusline()
  local function get_mode()
    local mode_map = {
      n = 'NORMAL',
      i = 'INSERT',
      v = 'VISUAL',
      V = 'V-LINE',
      ['␣'] = 'V-BLOCK',
      R = 'REPLACE',
      c = 'COMMAND',
      t = 'TERMINAL',
    }
    return mode_map[vim.fn.mode()] or 'UNKNOWN'
  end

  local function get_file_info()
    local filename = vim.fn.expand('%:t')
    if filename == '' then return '[No Name]' end
  end

```



```

local modified = vim.bo.modified and '[+]' or ''
local readonly = vim.bo.readonly and '[-]' or ''

return filename .. readonly .. modified
end

local function get_git_branch()
    local Job = require('plenary.job')
    local branch = ''

    Job:new({
        command = 'git',
        args = {'branch', '--show-current'},
        cwd = vim.fn.getcwd(),
        on_exit = function(j, return_val)
            if return_val == 0 then
                local result = j:result()
                if #result > 0 then
                    branch = ' ' .. result[1]
                end
            end
        end
    }):sync()

    return branch
end

local function get_lsp_status()
    local clients = vim.lsp.get_active_clients()
    if #clients > 0 then
        return ' LSP'
    end
    return ''
end

-- Build statusline
local mode = get_mode()
local file_info = get_file_info()
local git_branch = get_git_branch()
local lsp_status = get_lsp_status()
local line_col = vim.fn.line('.') .. ':' .. vim.fn.col('.')

return string.format('%#StatusLine# %s %#StatusLineNC# %s %#StatusLine# %s %s
%#StatusLineNC# %s ',
    mode, file_info, git_branch, lsp_status, line_col)
end

-- Set custom statusline
function _G.setup_custom_statusline()

```

```
vim.opt.statusline = '%!v:lua.custom_statusline()'
end
```

9. Trouble-inspired Diagnostic Viewer

```
-- Show all diagnostics in a quickfix-like window
function _G.show_diagnostics_list()
    local diagnostics = vim.diagnostic.get()
    local qf_list = {}

    for _, diagnostic in ipairs(diagnostics) do
        table.insert(qf_list, {
            filename = vim.api.nvim_buf_get_name(0),
            lnum = diagnostic.lnum + 1,
            col = diagnostic.col + 1,
            text = diagnostic.message,
            type = diagnostic.severity,
        })
    end

    if #qf_list > 0 then
        vim.fn.setqflist(qf_list)
        vim.cmd('copen')
    else
        vim.notify('No diagnostics found', vim.log.levels.INFO)
    end
end

-- Filter diagnostics by severity
function _G.filter_diagnostics(severity)
    local severity_map = {
        error = vim.diagnostic.severity.ERROR,
        warn = vim.diagnostic.severity.WARN,
        info = vim.diagnostic.severity.INFO,
        hint = vim.diagnostic.severity.HINT,
    }

    local diagnostics = vim.diagnostic.get(0, {
        severity = severity_map[severity]
    })

    local qf_list = {}
    for _, diagnostic in ipairs(diagnostics) do
        table.insert(qf_list, {
            filename = vim.api.nvim_buf_get_name(0),
            lnum = diagnostic.lnum + 1,
            col = diagnostic.col + 1,
            text = diagnostic.message,
        })
    end
end
```

```

if #qf_list > 0 then
    vim.fn.setqflist(qf_list)
    vim.cmd('copen')
else
    vim.notify('No ' .. severity .. ' diagnostics found', vim.log.levels.INFO)
end
end

```

Keybinding Suggestions:

```

-- Plugin-inspired keybindings
vim.keymap.set('n', '<leader>ff', '<cmd>lua fuzzy_find_files()<cr>')
vim.keymap.set('n', '<leader>fg', '<cmd>lua live_grep()<cr>')
vim.keymap.set('n', '<leader>?', '<cmd>lua show_keybindings()<cr>')
vim.keymap.set('n', '<leader>e', '<cmd>lua toggle_file_explorer()<cr>')
vim.keymap.set({'n', 'v'}, '<leader>/', '<cmd>lua toggle_comment()<cr>')
vim.keymap.set('n', '<leader>ig', '<cmd>lua toggle_indent_guides()<cr>')
vim.keymap.set('n', '<leader>gs', '<cmd>lua show_git_status()<cr>')
vim.keymap.set('n', '<leader>gb', '<cmd>lua show_git_blame()<cr>')
vim.keymap.set('n', '<leader>dd', '<cmd>lua show_diagnostics_list()<cr>')
vim.keymap.set('n', '<leader>de', '<cmd>lua filter_diagnostics("error")<cr>')
vim.keymap.set('n', '<leader>dw', '<cmd>lua filter_diagnostics("warn")<cr>')

-- Smart pairing (map these in insert mode)
vim.keymap.set('i', '(', '(<Left>', {noremap = true})
vim.keymap.set('i', '[', '[<Left>', {noremap = true})
vim.keymap.set('i', '{', '{<Left>', {noremap = true})
vim.keymap.set('i', '"', '"<Left>', {noremap = true})
vim.keymap.set('i', "'", "'<Left>", {noremap = true})

```

These functions replicate popular plugin functionality while keeping everything in pure Lua, giving you plugin-like features without external dependencies!
