with-qwen

guide me to create custom neovim plugin but using c-program as main backend

what if we want to build plugin mainly to enrich neovim apis functionality

explore another type of plugin and still using c-program as backend

inspire me even further with c-program as neovim plugin-backend

---

Creating a Custom Neovim Plugin with C Backend

I'll guide you through creating a Neovim plugin that uses a C program as the main backend. This approach involves creating a plugin that communicates between Neovim and your C program.

Architecture Overview

The typical architecture involves:

- A Lua plugin interface (frontend)
- A C program (backend)
- Communication via pipes, RPC, or shared memory

Step 1: Project Structure

```
my-nvim-plugin/
├── plugin/
│   └── myplugin.vim
├── lua/
│   └── myplugin/
│       ├── init.lua
│       └── backend.lua
├── src/
│   └── my_backend.c
├── Makefile
└── README.md
```

Step 2: C Backend Implementation

Create `src/my_backend.c` :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <json-c/json.h>

// Simple protocol: JSON over stdin/stdout
void process_request(const char* json_str) {
    json_object* obj = json_tokener_parse(json_str);
```

```
    if (!obj) return;

    json_object* method;
    if (!json_object_object_get_ex(obj, "method", &method)) {
        json_object_put(obj);
        return;
    }

    const char* method_str = json_object_get_string(method);

    // Example methods
    if (strcmp(method_str, "analyze") == 0) {
        json_object* response = json_object_new_object();
        json_object_object_add(response, "result", json_object_new_string("Analysis
complete"));
        json_object_object_add(response, "success", json_object_new_boolean(1));

        printf("%s\n", json_object_to_json_string(response));
        fflush(stdout);

        json_object_put(response);
    }
    else if (strcmp(method_str, "format") == 0) {
        json_object* code_obj;
        if (json_object_object_get_ex(obj, "code", &code_obj)) {
            const char* code = json_object_get_string(code_obj);

            // Your formatting logic here
            json_object* response = json_object_new_object();
            json_object_object_add(response, "formatted",
json_object_new_string(code));
            json_object_object_add(response, "success", json_object_new_boolean(1));

            printf("%s\n", json_object_to_json_string(response));
            fflush(stdout);

            json_object_put(response);
        }
    }

    json_object_put(obj);
}

int main() {
    char buffer[4096];

    // Signal that backend is ready
    printf("{\"ready\": true}\n");
    fflush(stdout);

    while (fgets(buffer, sizeof(buffer), stdin)) {
        size_t len = strlen(buffer);
```

```
        if (len > 0 && buffer[len-1] == '\n') {
            buffer[len-1] = '\0';
        }

        process_request(buffer);
    }

    return 0;
}
```

Step 3: Build System (Makefile)

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99 -O2
LIBS = -ljson-c

PLUGIN_NAME = myplugin
BACKEND_BIN = $(PLUGIN_NAME)_backend

# Installation paths
NVIM_DIR = ~/.local/share/nvim/site/pack/myplugins/start/$(PLUGIN_NAME)
BUILD_DIR = build

.PHONY: all clean install

all: $(BUILD_DIR)/$(BACKEND_BIN)

$(BUILD_DIR):
    mkdir -p $(BUILD_DIR)

$(BUILD_DIR)/$(BACKEND_BIN): src/my_backend.c | $(BUILD_DIR)
    $(CC) $(CFLAGS) -o $@ $< $(LIBS)

clean:
    rm -rf $(BUILD_DIR)

install: $(BUILD_DIR)/$(BACKEND_BIN)
    cp -r . $(NVIM_DIR)
    cp $(BUILD_DIR)/$(BACKEND_BIN) $(NVIM_DIR)/

uninstall:
    rm -rf $(NVIM_DIR)
```

Step 4: Lua Plugin Interface

Create `lua/myplugin/init.lua`:

```
local M = {}
```

```lua
-- Import backend communication module
local backend = require('myplugin.backend')

function M.setup()
    -- Initialize the C backend
    backend.start_backend()
end

function M.analyze_current_buffer()
    local current_buf = vim.api.nvim_get_current_buf()
    local lines = vim.api.nvim_buf_get_lines(current_buf, 0, -1, false)
    local content = table.concat(lines, '\n')

    local params = {
        method = 'analyze',
        buffer = vim.api.nvim_buf_get_name(current_buf),
        content = content
    }

    backend.send_request(params, function(result)
        if result.success then
            vim.notify('Analysis complete: ' .. result.result, vim.log.levels.INFO)
        else
            vim.notify('Analysis failed', vim.log.levels.ERROR)
        end
    end)
end

function M.format_current_buffer()
    local current_buf = vim.api.nvim_get_current_buf()
    local lines = vim.api.nvim_buf_get_lines(current_buf, 0, -1, false)
    local content = table.concat(lines, '\n')

    local params = {
        method = 'format',
        code = content
    }

    backend.send_request(params, function(result)
        if result.success then
            vim.api.nvim_buf_set_lines(current_buf, 0, -1, false,
vim.split(result.formatted, '\n'))
            vim.notify('Buffer formatted successfully', vim.log.levels.INFO)
        else
            vim.notify('Formatting failed', vim.log.levels.ERROR)
        end
    end)
end

return M
```

Step 5: Backend Communication Module

Create `lua/myplugin/backend.lua`:

```
local M = {}

local Job = require('plenary.job')

local backend_process = nil
local pending_requests = {}
local request_id = 0

function M.start_backend()
    local backend_path = vim.fn.fnamemodify(debug.getinfo(1).source:match('@?(.*)'),
':p:h:h') .. '/build/myplugin_backend'

    backend_process = Job:new({
        command = backend_path,
        on_stdout = function(_, data)
            if data and match('"ready": true') then
                print('C backend started successfully')
            else
                -- Handle responses
                local success, result = pcall(vim.json.decode, data)
                if success and result then
                    local id = result.id
                    if id and pending_requests[id] then
                        pending_requests[id].callback(result)
                        pending_requests[id] = nil
                    end
                end
            end
        end,
        on_stderr = function(_, data)
            if data then
                print('Backend error: ' .. data)
            end
        end,
        on_exit = function()
            print('C backend exited')
            backend_process = nil
        end,
    })

    backend_process:start()
end

function M.send_request(params, callback)
    if not backend_process then
        vim.notify('Backend not running', vim.log.levels.ERROR)
        return
```

```
    end

    request_id = request_id + 1
    local req_id = request_id

    params.id = req_id
    pending_requests[req_id] = { callback = callback }

    local json_str = vim.json.encode(params)
    backend_process:send(json_str .. '\n')
end

function M.stop_backend()
    if backend_process then
        backend_process:shutdown()
        backend_process = nil
    end
end

return M
```

Step 6: Vim Plugin File

Create `plugin/myplugin.vim`:

```
if exists('g:loaded_myplugin')
  finish
endif
let g:loaded_myplugin = 1

" Setup function
command! MyPluginSetup lua require('myplugin').setup()

" Commands
command! MyPluginAnalyze lua require('myplugin').analyze_current_buffer()
command! MyPluginFormat lua require('myplugin').format_current_buffer()

" Key mappings
nnoremap <leader>ma :MyPluginAnalyze<CR>
nnoremap <leader>mf :MyPluginFormat<CR>
```

Step 7: Installation and Usage

1. **Build the plugin:**

```
make
make install
```

2. **Add to your Neovim config:**

```lua
-- In your init.lua
require('myplugin').setup()
```

3. **Use the plugin:**

- `:MyPluginAnalyze` - Analyze current buffer
- `:MyPluginFormat` - Format current buffer
- `<leader>ma` - Analyze shortcut
- `<leader>mf` - Format shortcut

Advanced Communication Options

For more robust communication, consider:

Option A: Using Neovim's Job API with RPC

```lua
-- More sophisticated RPC implementation
local uv = vim.loop

local function create_rpc_connection()
    local handle, pid = uv.spawn('my_backend', {
        stdio = { 'pipe', 'pipe', 'pipe' }
    }, function(code, signal)
        print('Backend exited', code, signal)
    end)

    -- Set up communication
    local stdin = handle:stdio_stream(0)
    local stdout = handle:stdio_stream(1)

    -- Handle responses
    stdout:read_start(function(err, chunk)
        if chunk then
            -- Process chunk
        end
    end)

    return {
        send = function(data)
            stdin:write(data .. '\n')
        end,
        close = function()
            handle:close()
        end
    }
end
```

Option B: Using Unix Sockets

```c
// In your C backend, you can also support Unix sockets
#include <sys/socket.h>
#include <sys/un.h>

int create_socket_server(const char* socket_path) {
    int sock = socket(AF_UNIX, SOCK_STREAM, 0);
    struct sockaddr_un addr;

    memset(&addr, 0, sizeof(addr));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, socket_path, sizeof(addr.sun_path) - 1);

    bind(sock, (struct sockaddr*)&addr, sizeof(addr));
    listen(sock, 1);

    return sock;
}
```

Best Practices

1. **Error Handling**: Always handle communication failures
2. **Memory Management**: Properly free C resources
3. **Security**: Validate all input from Neovim
4. **Performance**: Consider caching and async operations
5. **Logging**: Add logging for debugging communication issues

This setup gives you a solid foundation for a Neovim plugin with a C backend, allowing you to leverage C's performance while maintaining Neovim's extensibility through Lua.

Creating a Neovim Plugin to Enhance Core APIs

When building a plugin to enrich Neovim's core APIs, you're essentially creating extensions that augment existing functionality or add new capabilities to the editor. Here's how to approach this:

Architecture Overview

```
enhanced-nvim/
├── plugin/
│   └── enhanced_nvim.vim
├── lua/
│   └── enhanced_nvim/
│       ├── init.lua
│       ├── buffer.lua
│       ├── window.lua
│       ├── tree_sitter.lua
│       └── extensions.lua
├── src/
│   └── core_extensions.c
```

```
├── Makefile
└── README.md
```

Step 1: Enhanced Buffer Management

Create `lua/enhanced_nvim/buffer.lua`:

```lua
local M = {}

-- Enhanced buffer operations
function M.create_scratch_buffer(name, content)
    local buf = vim.api.nvim_create_buf(false, true) -- not listed, scratch
    vim.api.nvim_buf_set_option(buf, 'buftype', 'nofile')
    vim.api.nvim_buf_set_option(buf, 'bufhidden', 'hide')
    vim.api.nvim_buf_set_option(buf, 'swapfile', false)

    if name then
        vim.api.nvim_buf_set_name(buf, name)
    end

    if content then
        vim.api.nvim_buf_set_lines(buf, 0, -1, false, vim.split(content, '\n'))
    end

    return buf
end

function M.get_buffer_stats(bufnr)
    bufnr = bufnr or 0
    local lines = vim.api.nvim_buf_get_lines(bufnr, 0, -1, false)
    local stats = {
        line_count = #lines,
        char_count = 0,
        word_count = 0,
        longest_line = 0
    }

    for _, line in ipairs(lines) do
        stats.char_count = stats.char_count + #line
        stats.word_count = stats.word_count + #vim.split(line, '%s+')
        stats.longest_line = math.max(stats.longest_line, #line)
    end

    return stats
end

function M.buffer_fuzzy_find(pattern)
    local buffers = vim.api.nvim_list_bufs()
    local matches = {}

    for _, buf in ipairs(buffers) do
```

```
        if vim.api.nvim_buf_is_loaded(buf) then
            local name = vim.api.nvim_buf_get_name(buf)
            if string.find(name:lower(), pattern:lower()) then
                table.insert(matches, {buf = buf, name = name})
            end
        end
    end

    return matches
end

return M
```

Step 2: Window and Tab Extensions

Create `lua/enhanced_nvim/window.lua`:

```
local M = {}

-- Enhanced window management
function M.create_floating_window(opts)
    opts = opts or {}
    local width = opts.width or 80
    local height = opts.height or 20
    local row = opts.row or math.floor((vim.o.lines - height) / 2)
    local col = opts.col or math.floor((vim.o.columns - width) / 2)

    local buf = vim.api.nvim_create_buf(false, true)
    local win = vim.api.nvim_open_win(buf, true, {
        relative = opts.relative or 'editor',
        width = width,
        height = height,
        row = row,
        col = col,
        style = opts.style or 'minimal',
        border = opts.border or 'rounded',
        zindex = opts.zindex or 50
    })

    -- Enhanced floating window features
    if opts.auto_close then
        vim.api.nvim_create_autocmd('BufLeave', {
            buffer = buf,
            callback = function()
                if vim.api.nvim_win_is_valid(win) then
                    vim.api.nvim_win_close(win, true)
                end
            end
        })
    end
```

```
    return {win = win, buf = buf}
end

function M.swap_windows(win1, win2)
    win1 = win1 or vim.api.nvim_get_current_win()
    win2 = win2 or vim.fn.win_getid(vim.fn.winnr('#'))

    local buf1 = vim.api.nvim_win_get_buf(win1)
    local buf2 = vim.api.nvim_win_get_buf(win2)

    vim.api.nvim_win_set_buf(win1, buf2)
    vim.api.nvim_win_set_buf(win2, buf1)
end

function M.focus_window_with_buffer(bufnr)
    local wins = vim.api.nvim_list_wins()
    for _, win in ipairs(wins) do
        if vim.api.nvim_win_get_buf(win) == bufnr then
            vim.api.nvim_set_current_win(win)
            return true
        end
    end
    return false
end

return M
```

Step 3: Tree-sitter Extensions

Create `lua/enhanced_nvim/tree_sitter.lua`:

```
local M = {}

function M.get_node_info_at_cursor()
    local ts_utils = require('nvim-treesitter.ts_utils')
    local node = ts_utils.get_node_at_cursor()

    if not node then return nil end

    local start_row, start_col, end_row, end_col = node:range()

    return {
        node = node,
        type = node:type(),
        start_pos = {start_row, start_col},
        end_pos = {end_row, end_col},
        text = vim.treesitter.get_node_text(node, 0),
        parent = node:parent(),
        children = {node:child_count()}
    }
end
```

```lua
function M.select_enclosing_function()
    local ts_utils = require('nvim-treesitter.ts_utils')
    local parsers = require('nvim-treesitter.parsers')

    if not parsers.has_parser() then return end

    local cursor = vim.api.nvim_win_get_cursor(0)
    local current_node = ts_utils.get_node_at_cursor()

    while current_node do
        local node_type = current_node:type()
        if string.match(node_type, 'function') or string.match(node_type, 'method')
then
            ts_utils.select_node(current_node)
            return
        end
        current_node = current_node:parent()
    end
end

function M.extract_function_signature()
    local node_info = M.get_node_info_at_cursor()
    if not node_info then return nil end

    local function get_function_signature(node)
        -- This is language-specific - you'd need to implement per language
        local lang = vim.bo.filetype
        local query = vim.treesitter.query.get(lang, 'locals')

        if query then
            for id, node in query:iter_captures(node, 0, 0, -1) do
                local name = query.captures[id]
                if name == 'function' then
                    return vim.treesitter.get_node_text(node, 0)
                end
            end
        end
        return nil
    end

    return get_function_signature(node_info.node)
end

return M
```

Step 4: Core API Extensions

Create `lua/enhanced_nvim/extensions.lua`:

```lua
local M = {}

-- Enhanced search functionality
function M.search_in_files(pattern, opts)
    opts = opts or {}
    local search_dir = opts.dir or vim.fn.expand('%:p:h')
    local file_pattern = opts.files or '*'

    local cmd = string.format('rg --json "%s" %s', pattern, search_dir)
    local result = vim.fn.system(cmd)

    local matches = {}
    for line in result:gmatch("[^\r\n]+") do
        local success, data = pcall(vim.json.decode, line)
        if success and data.type == 'match' then
            table.insert(matches, {
                filename = data.data.path.text,
                line = data.data.line_number,
                content = data.data.lines.text,
                column = data.data.column
            })
        end
    end

    return matches
end

-- Enhanced undo/redo with visual feedback
function M.visual_undo_redo()
    local current_pos = vim.api.nvim_win_get_cursor(0)
    local success = pcall(vim.cmd, 'undo')

    if success then
        vim.api.nvim_win_set_cursor(0, current_pos)
        -- Highlight the changed area
        local new_pos = vim.api.nvim_win_get_cursor(0)
        vim.highlight.range(
            0,
            'Visual',
            'a',
            {new_pos[1] - 2, 0},
            {new_pos[1], -1}
        )
    end
end

-- Enhanced yank/paste with history
function M.yank_with_history()
    local text = vim.fn.getreg('"')
    if text and text ~= '' then
        local history = vim.g.yank_history or {}
```

```
        table.insert(history, 1, text)
        -- Keep only last 10 entries
        if #history > 10 then
            history = {table.unpack(history, 1, 10)}
        end
        vim.g.yank_history = history
    end
end

-- Enhanced file operations
function M.safe_write_file(filename, content)
    filename = filename or vim.api.nvim_buf_get_name(0)
    if filename == '' then
        vim.notify('No filename specified', vim.log.levels.ERROR)
        return false
    end

    -- Create backup
    local backup_name = filename .. '.bak'
    if vim.fn.filereadable(filename) == 1 then
        vim.fn.writefile(vim.fn.readfile(filename), backup_name)
    end

    -- Write new content
    local success, err = pcall(vim.fn.writefile, vim.split(content, '\n'), filename)
    if not success then
        vim.notify('Failed to write file: ' .. (err or 'unknown error'),
vim.log.levels.ERROR)
        return false
    end

    return true
end

return M
```

Step 5: Main Plugin Entry Point

Create `lua/enhanced_nvim/init.lua`:

```
local M = {}

-- Lazy load modules
local modules = {
    buffer = nil,
    window = nil,
    tree_sitter = nil,
    extensions = nil
}

local function lazy_load(module_name)
```

```lua
    if not modules[module_name] then
        modules[module_name] = require('enhanced_nvim.' .. module_name)
    end
    return modules[module_name]
end

-- Buffer extensions
function M.create_scratch_buffer(name, content)
    return lazy_load('buffer').create_scratch_buffer(name, content)
end

function M.get_buffer_stats(bufnr)
    return lazy_load('buffer').get_buffer_stats(bufnr)
end

-- Window extensions
function M.create_floating_window(opts)
    return lazy_load('window').create_floating_window(opts)
end

function M.swap_windows(win1, win2)
    return lazy_load('window').swap_windows(win1, win2)
end

-- Tree-sitter extensions
function M.get_node_info_at_cursor()
    return lazy_load('tree_sitter').get_node_info_at_cursor()
end

function M.select_enclosing_function()
    return lazy_load('tree_sitter').select_enclosing_function()
end

-- General extensions
function M.search_in_files(pattern, opts)
    return lazy_load('extensions').search_in_files(pattern, opts)
end

function M.safe_write_file(filename, content)
    return lazy_load('extensions').safe_write_file(filename, content)
end

-- Setup function
function M.setup(opts)
    opts = opts or {}

    -- Set up key mappings
    vim.api.nvim_set_keymap('n', '<leader>sf',
        ':lua require("enhanced_nvim").select_enclosing_function()<CR>',
        { noremap = true, silent = true })

    vim.api.nvim_set_keymap('n', '<leader>ss',
```

```
        ':lua require("enhanced_nvim").get_node_info_at_cursor()<CR>',
        { noremap = true, silent = true })

    -- Set up autocommands for enhanced functionality
    vim.api.nvim_create_autocmd('TextYankPost', {
        callback = function()
            require('enhanced_nvim.extensions').yank_with_history()
        end,
        desc = 'Enhanced yank history'
    })

    -- Override some default behaviors if needed
    if opts.enhanced_write then
        vim.api.nvim_create_autocmd('BufWritePre', {
            callback = function()
                -- Custom write logic here
            end
        })
    end
end

return M
```

Step 6: Vim Plugin File

Create `plugin/enhanced_nvim.vim`:

```
if exists('g:loaded_enhanced_nvim')
  finish
endif
let g:loaded_enhanced_nvim = 1

" Commands
command! -nargs=1 ScratchBuffer lua
require('enhanced_nvim').create_scratch_buffer(<f-args>)
command! BufferStats lua
print(vim.inspect(require('enhanced_nvim').get_buffer_stats()))
command! -nargs=1 SearchFiles lua
print(vim.inspect(require('enhanced_nvim').search_in_files(<f-args>)))

" Key mappings
nnoremap <silent> <leader>bs :lua require('enhanced_nvim').get_buffer_stats()<CR>
nnoremap <silent> <leader>sf :lua
require('enhanced_nvim').select_enclosing_function()<CR>
nnoremap <silent> <leader>ss :lua require('enhanced_nvim').get_node_info_at_cursor()
<CR>

" Auto-commands
augroup EnhancedNvim
  autocmd!
```

```
  autocmd FileType * lua require('enhanced_nvim').setup()
augroup END
```

Step 7: Advanced C Backend for Performance-Critical Operations

Create `src/core_extensions.c` for performance-intensive operations:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <json-c/json.h>

// Fast text processing functions
typedef struct {
    size_t lines;
    size_t chars;
    size_t words;
    size_t longest_line;
} text_stats_t;

text_stats_t analyze_text(const char* text) {
    text_stats_t stats = {0, 0, 0, 0};
    const char* ptr = text;

    while (*ptr) {
        stats.chars++;

        // Count words (simple whitespace-based)
        if (isspace(*ptr)) {
            while (isspace(*ptr)) ptr++;
            if (*ptr) stats.words++;
        } else {
            ptr++;
        }
    }

    // Count lines and find longest
    ptr = text;
    size_t current_line = 0;
    while (*ptr) {
        if (*ptr == '\n') {
            stats.lines++;
            if (current_line > stats.longest_line) {
                stats.longest_line = current_line;
            }
            current_line = 0;
        } else {
            current_line++;
        }
        ptr++;
```

```
    }

    return stats;
}

// Fast pattern matching
int fast_pattern_match(const char* text, const char* pattern) {
    // Implement Boyer-Moore or similar for fast searching
    const char* pos = strstr(text, pattern);
    return pos ? (int)(pos - text) : -1;
}

int main() {
    char buffer[8192];
    while (fgets(buffer, sizeof(buffer), stdin)) {
        json_object* request = json_tokener_parse(buffer);
        if (!request) continue;

        json_object* method;
        if (!json_object_object_get_ex(request, "method", &method)) {
            json_object_put(request);
            continue;
        }

        const char* method_str = json_object_get_string(method);

        if (strcmp(method_str, "analyze_text") == 0) {
            json_object* text_obj;
            if (json_object_object_get_ex(request, "text", &text_obj)) {
                const char* text = json_object_get_string(text_obj);
                text_stats_t stats = analyze_text(text);

                json_object* response = json_object_new_object();
                json_object_object_add(response, "lines",
json_object_new_int64(stats.lines));
                json_object_object_add(response, "chars",
json_object_new_int64(stats.chars));
                json_object_object_add(response, "words",
json_object_new_int64(stats.words));
                json_object_object_add(response, "longest_line",
json_object_new_int64(stats.longest_line));

                printf("%s\n", json_object_to_json_string(response));
                fflush(stdout);
                json_object_put(response);
            }
        }

        json_object_put(request);
    }
```

```
    return 0;
}
```

Step 8: Integration with Existing APIs

To truly enhance Neovim's APIs, you can monkey-patch or extend existing functions:

```
-- In your init.lua, you can extend existing API functions
local original_buf_set_lines = vim.api.nvim_buf_set_lines

function vim.api.nvim_buf_set_lines(buf, start, end_, strict_indexing, replacement)
    -- Add your enhancement logic here
    local success, result = pcall(original_buf_set_lines, buf, start, end_,
strict_indexing, replacement)

    if success then
        -- Trigger enhanced events
        vim.api.nvim_exec_autocmds('User', {
            pattern = 'EnhancedBufSetLines',
            modeline = false,
            data = {
                buffer = buf,
                start = start,
                replacement = replacement
            }
        })
    end

    return result
end
```

This approach gives you a comprehensive framework for enhancing Neovim's core functionality while maintaining compatibility with existing APIs and providing new, powerful extensions.

Creating a Language Server Protocol (LSP) Plugin with C Backend

Let's explore a **Language Server Protocol (LSP) plugin** that uses a C program as the backend. This is particularly useful for creating LSP servers for languages that don't have good support or for implementing custom analysis tools.

Architecture Overview

```
c-lsp-plugin/
├── plugin/
│   └── clsp.vim
├── lua/
```

```
│   └── clsp/
│       ├── init.lua
│       ├── server.lua
│       ├── client.lua
│       └── handlers.lua
├── src/
│   ├── lsp_server.c
│   ├── json_parser.c
│   └── language_analyzer.c
├── Makefile
└── README.md
```

Step 1: C Language Server Implementation

Create `src/lsp_server.c`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <json-c/json.h>

// LSP Protocol structures
typedef struct {
    char* uri;
    char* text;
    int version;
} document_t;

typedef struct {
    int line;
    int character;
} position_t;

typedef struct {
    position_t start;
    position_t end;
} range_t;

// Global document storage
document_t documents[100];
int document_count = 0;

// Helper functions
char* read_content_length(const char* header) {
    const char* content_len = strstr(header, "Content-Length: ");
    if (!content_len) return NULL;

    content_len += 16; // Length of "Content-Length: "
    const char* end = strchr(content_len, '\r');
    if (!end) return NULL;
```

```
    int len = atoi(content_len);
    return strndup(content_len, end - content_len);
}

void send_response(const char* method, json_object* params) {
    json_object* response = json_object_new_object();
    json_object_object_add(response, "jsonrpc", json_object_new_string("2.0"));
    json_object_object_add(response, "method", json_object_new_string(method));
    json_object_object_add(response, "params", params);

    const char* json_str = json_object_to_json_string(response);
    int len = strlen(json_str);

    printf("Content-Length: %d\r\n\r\n%s", len, json_str);
    fflush(stdout);

    json_object_put(response);
}

void handle_initialize(json_object* params) {
    json_object* capabilities = json_object_new_object();

    // Server capabilities
    json_object* text_doc_caps = json_object_new_object();
    json_object_object_add(text_doc_caps, "definitionProvider",
json_object_new_boolean(1));
    json_object_object_add(text_doc_caps, "hoverProvider",
json_object_new_boolean(1));
    json_object_object_add(text_doc_caps, "documentSymbolProvider",
json_object_new_boolean(1));
    json_object_object_add(text_doc_caps, "completionProvider",
json_object_new_object());
    json_object_object_add(text_doc_caps, "codeActionProvider",
json_object_new_boolean(1));

    json_object_object_add(capabilities, "textDocument", text_doc_caps);
    json_object_object_add(capabilities, "workspace", json_object_new_object());

    json_object* result = json_object_new_object();
    json_object_object_add(result, "capabilities", capabilities);
    json_object_object_add(result, "serverInfo", json_object_new_object());

    send_response("result", result);
}

void handle_text_document_did_open(json_object* params) {
    json_object* text_doc;
    if (json_object_object_get_ex(params, "textDocument", &text_doc)) {
        json_object* uri_obj, *text_obj, *version_obj;

        if (json_object_object_get_ex(text_doc, "uri", &uri_obj) &&
```

```
            json_object_object_get_ex(text_doc, "text", &text_obj) &&
            json_object_object_get_ex(text_doc, "version", &version_obj)) {

            documents[document_count].uri = strdup(json_object_get_string(uri_obj));
            documents[document_count].text =
strdup(json_object_get_string(text_obj));
            documents[document_count].version = json_object_get_int(version_obj);
            document_count++;
        }
    }
}

void handle_text_document_completion(json_object* params) {
    // Simple completion logic
    json_object* items = json_object_new_array();

    // Add some sample completions
    json_object* item1 = json_object_new_object();
    json_object_object_add(item1, "label", json_object_new_string("printf"));
    json_object_object_add(item1, "kind", json_object_new_int(3)); // Function
    json_object_array_add(items, item1);

    json_object* item2 = json_object_new_object();
    json_object_object_add(item2, "label", json_object_new_string("scanf"));
    json_object_object_add(item2, "kind", json_object_new_int(3)); // Function
    json_object_array_add(items, item2);

    json_object* result = json_object_new_object();
    json_object_object_add(result, "items", items);

    send_response("result", result);
}

void handle_request(const char* json_str) {
    json_object* request = json_tokener_parse(json_str);
    if (!request) return;

    json_object* method_obj;
    if (!json_object_object_get_ex(request, "method", &method_obj)) {
        json_object_put(request);
        return;
    }

    const char* method = json_object_get_string(method_obj);

    if (strcmp(method, "initialize") == 0) {
        handle_initialize(request);
    } else if (strcmp(method, "textDocument/didOpen") == 0) {
        handle_text_document_did_open(request);
    } else if (strcmp(method, "textDocument/completion") == 0) {
        handle_text_document_completion(request);
    } else if (strcmp(method, "shutdown") == 0) {
```

```
        printf("Content-Length: 13\r\n\r\n{\"result\":null}");
        fflush(stdout);
    }

    json_object_put(request);
}

int main() {
    char buffer[8192];

    // LSP protocol uses Content-Length headers
    while (1) {
        char header[1024] = {0};
        char* line;

        // Read headers until \r\n\r\n
        while ((line = fgets(buffer, sizeof(buffer), stdin)) != NULL) {
            if (strcmp(buffer, "\r\n") == 0) break;
            strcat(header, buffer);
        }

        // Get content length
        char* len_str = read_content_length(header);
        if (!len_str) continue;

        int content_len = atoi(len_str);
        free(len_str);

        // Read the JSON content
        char* content = malloc(content_len + 1);
        fread(content, 1, content_len, stdin);
        content[content_len] = '\0';

        handle_request(content);
        free(content);
    }

    return 0;
}
```

Step 2: Language Analysis Module

Create `src/language_analyzer.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <json-c/json.h>

// Token types
typedef enum {
```

```
    TOKEN_KEYWORD,
    TOKEN_IDENTIFIER,
    TOKEN_NUMBER,
    TOKEN_STRING,
    TOKEN_OPERATOR,
    TOKEN_PUNCTUATION
} token_type_t;

typedef struct {
    token_type_t type;
    char* value;
    int line;
    int column;
} token_t;

// Simple C language tokenizer
token_t* tokenize_c_code(const char* code, int* token_count) {
    const char* keywords[] = {"int", "char", "float", "double", "void", "if",
"else",
                              "while", "for", "return", "struct", "typedef", NULL};

    token_t* tokens = malloc(1000 * sizeof(token_t)); // Max 1000 tokens
    *token_count = 0;

    const char* ptr = code;
    int line = 1, col = 1;

    while (*ptr) {
        // Skip whitespace
        while (*ptr == ' ' || *ptr == '\t' || *ptr == '\n') {
            if (*ptr == '\n') { line++; col = 1; }
            else { col++; }
            ptr++;
        }

        if (!*ptr) break;

        // Check for keywords
        int is_keyword = 0;
        for (int i = 0; keywords[i]; i++) {
            int len = strlen(keywords[i]);
            if (strncmp(ptr, keywords[i], len) == 0 &&
                !isalnum(ptr[len]) && ptr[len] != '_') {
                tokens[*token_count].type = TOKEN_KEYWORD;
                tokens[*token_count].value = strndup(ptr, len);
                tokens[*token_count].line = line;
                tokens[*token_count].column = col;
                (*token_count)++;
                ptr += len;
                col += len;
                is_keyword = 1;
                break;
```

```
    }
}

if (is_keyword) continue;

// Handle identifiers
if (isalpha(*ptr) || *ptr == '_') {
    const char* start = ptr;
    int start_col = col;
    while (isalnum(*ptr) || *ptr == '_') { ptr++; col++; }

    tokens[*token_count].type = TOKEN_IDENTIFIER;
    tokens[*token_count].value = strndup(start, ptr - start);
    tokens[*token_count].line = line;
    tokens[*token_count].column = start_col;
    (*token_count)++;
    continue;
}

// Handle numbers
if (isdigit(*ptr)) {
    const char* start = ptr;
    int start_col = col;
    while (isdigit(*ptr) || *ptr == '.') { ptr++; col++; }

    tokens[*token_count].type = TOKEN_NUMBER;
    tokens[*token_count].value = strndup(start, ptr - start);
    tokens[*token_count].line = line;
    tokens[*token_count].column = start_col;
    (*token_count)++;
    continue;
}

// Handle strings
if (*ptr == '"' || *ptr == '\'') {
    char quote = *ptr;
    const char* start = ptr;
    int start_col = col;
    ptr++; col++;

    while (*ptr && *ptr != quote) {
        if (*ptr == '\\' && ptr[1]) { ptr++; col++; } // Escape sequence
        ptr++; col++;
    }

    if (*ptr == quote) { ptr++; col++; } // Include closing quote

    tokens[*token_count].type = TOKEN_STRING;
    tokens[*token_count].value = strndup(start, ptr - start);
    tokens[*token_count].line = line;
    tokens[*token_count].column = start_col;
    (*token_count)++;
```

```
            continue;
        }

        // Handle operators and punctuation
        char op[2] = {*ptr, '\0'};
        tokens[*token_count].type = TOKEN_OPERATOR;
        tokens[*token_count].value = strdup(op);
        tokens[*token_count].line = line;
        tokens[*token_count].column = col;
        (*token_count)++;

        ptr++;
        col++;
    }

    return tokens;
}

// Generate symbols from tokens
json_object* generate_symbols_from_tokens(token_t* tokens, int count) {
    json_object* symbols = json_object_new_array();

    for (int i = 0; i < count; i++) {
        if (tokens[i].type == TOKEN_IDENTIFIER) {
            // Look for function definitions (identifier followed by '(')
            if (i + 1 < count && strcmp(tokens[i + 1].value, "(") == 0) {
                json_object* symbol = json_object_new_object();
                json_object_object_add(symbol, "name",
json_object_new_string(tokens[i].value));
                json_object_object_add(symbol, "kind", json_object_new_int(12)); //
Function
                json_object_object_add(symbol, "location",
json_object_new_object());
                json_object_array_add(symbols, symbol);
            }
        }
    }

    return symbols;
}
```

Step 3: Lua LSP Client

Create `lua/clsp/client.lua`:

```
local M = {}

local Job = require('plenary.job')

function M.start_lsp_server()
    local server_path = vim.fn.fnamemodify(debug.getinfo(1).source:match('@?(.*)'),
```

```
':p:h:h') .. '/build/clsp_server'

    local server_job = Job:new({
        command = server_path,
        on_stdout = function(_, data)
            if data then
                -- Parse LSP response
                local response = M.parse_lsp_response(data)
                if response then
                    M.handle_lsp_response(response)
                end
            end
        end,
        on_stderr = function(_, data)
            if data then
                print('LSP Server Error: ' .. data)
            end
        end,
        on_exit = function()
            print('LSP Server exited')
        end,
    })

    server_job:start()
    return server_job
end

function M.parse_lsp_response(data)
    -- Parse Content-Length header and JSON body
    local header_end = find('\r\n\r\n')
    if not header_end then return nil end

    local json_str = data:sub(header_end + 4)
    local success, parsed = pcall(vim.json.decode, json_str)
    return success and parsed or nil
end

function M.handle_lsp_response(response)
    if response.method then
        -- Handle server-to-client notifications
        if response.method == 'window/showMessage' then
            vim.notify(response.params.message, vim.log.levels.INFO)
        end
    elseif response.id then
        -- Handle responses to our requests
        M.handle_response(response)
    end
end

function M.send_lsp_request(method, params, id)
    local request = {
        jsonrpc = '2.0',
```

```
        method = method,
        params = params,
        id = id
    }

    local json_str = vim.json.encode(request)
    local msg = string.format('Content-Length: %d\r\n\r\n%s', #json_str, json_str)

    -- Send to server (you'd need to maintain the server connection)
    print('Sending LSP request: ' .. msg)
end

return M
```

Step 4: LSP Server Management

Create `lua/clsp/server.lua`:

```
local M = {}

local lsp_client = require('clsp.client')
local server_process = nil

function M.setup_server()
    if server_process then
        print('LSP server already running')
        return
    end

    server_process = lsp_client.start_lsp_server()

    -- Send initialize request
    local initialize_params = {
        processId = vim.fn.getpid(),
        rootUri = vim.uri_from_fname(vim.fn.getcwd()),
        capabilities = {
            textDocument = {
                completion = {
                    completionItem = {
                        snippetSupport = false
                    }
                },
                hover = { dynamicRegistration = true },
                definition = { dynamicRegistration = true },
                documentSymbol = { dynamicRegistration = true }
            }
        }
    }

    lsp_client.send_lsp_request('initialize', initialize_params, 1)
    lsp_client.send_lsp_request('initialized', {}, nil)
```

```
end

function M.shutdown_server()
    if server_process then
        lsp_client.send_lsp_request('shutdown', {}, 2)
        server_process:shutdown()
        server_process = nil
    end
end

function M.on_buffer_open(bufnr)
    local filename = vim.api.nvim_buf_get_name(bufnr)
    local content = table.concat(vim.api.nvim_buf_get_lines(bufnr, 0, -1, false),
'\n')

    local params = {
        textDocument = {
            uri = vim.uri_from_fname(filename),
            languageId = vim.bo[bufnr].filetype,
            version = 1,
            text = content
        }
    }

    lsp_client.send_lsp_request('textDocument/didOpen', params, nil)
end

function M.request_completion(bufnr, line, col)
    local params = {
        textDocument = {
            uri = vim.uri_from_fname(vim.api.nvim_buf_get_name(bufnr))
        },
        position = {
            line = line,
            character = col
        }
    }

    lsp_client.send_lsp_request('textDocument/completion', params, 3)
end

return M
```

Step 5: Main Plugin Initialization

Create `lua/clsp/init.lua`:

```
local M = {}

local server = require('clsp.server')
```

```lua
function M.setup(opts)
    opts = opts or {}

    -- Start the LSP server
    server.setup_server()

    -- Set up autocommands for file operations
    vim.api.nvim_create_autocmd('BufReadPost', {
        callback = function(args)
            server.on_buffer_open(args.buf)
        end,
        desc = 'Send didOpen notification to C LSP server'
    })

    -- Set up key mappings for LSP features
    vim.api.nvim_set_keymap('n', 'gD',
        ':lua require("clsp").goto_definition()<CR>',
        { noremap = true, silent = true })

    vim.api.nvim_set_keymap('n', 'K',
        ':lua require("clsp").hover()<CR>',
        { noremap = true, silent = true })

    vim.api.nvim_set_keymap('n', '<leader>ca',
        ':lua require("clsp").code_action()<CR>',
        { noremap = true, silent = true })
end

function M.goto_definition()
    local bufnr = vim.api.nvim_get_current_buf()
    local cursor = vim.api.nvim_win_get_cursor(0)

    -- In a real implementation, you'd send a request to your C server
    print('Requesting definition at line ' .. cursor[1] .. ', col ' .. cursor[2])
end

function M.hover()
    local bufnr = vim.api.nvim_get_current_buf()
    local cursor = vim.api.nvim_win_get_cursor(0)

    -- In a real implementation, you'd send a request to your C server
    print('Requesting hover information at line ' .. cursor[1] .. ', col ' ..
cursor[2])
end

function M.code_action()
    local bufnr = vim.api.nvim_get_current_buf()
    local cursor = vim.api.nvim_win_get_cursor(0)

    -- In a real implementation, you'd send a request to your C server
    print('Requesting code actions at line ' .. cursor[1] .. ', col ' .. cursor[2])
end
```

```lua
function M.on_attach(client, bufnr)
    -- This would be called by the built-in LSP client
    print('C LSP server attached to buffer ' .. bufnr)
end

return M
```

Step 6: Plugin Integration with Built-in LSP

Create `lua/clsp/handlers.lua`:

```lua
local M = {}

-- Custom handlers for LSP responses
function M.handle_completion_response(result)
    if result and result.items then
        -- Process completion items
        local items = {}
        for _, item in ipairs(result.items) do
            table.insert(items, {
                word = item.label,
                kind = item.kind and M.completion_kinds[item.kind] or '',
                menu = item.detail or '',
                info = item.documentation and item.documentation.value or ''
            })
        end

        -- Use vim's built-in completion
        vim.fn.complete(1, items)
    end
end

function M.handle_hover_response(result)
    if result and result.contents then
        -- Show hover information in floating window
        local content = result.contents.value or result.contents
        if type(content) == 'table' then
            content = table.concat(content, '\n')
        end

        local win = require('clsp.server').show_floating_window(content)
        return win
    end
end

M.completion_kinds = {
    [1] = 'Text',
    [2] = 'Method',
    [3] = 'Function',
    [4] = 'Constructor',
```

```
    [5] = 'Field',
    [6] = 'Variable',
    [7] = 'Class',
    [8] = 'Interface',
    [9] = 'Module',
    [10] = 'Property',
    [11] = 'Unit',
    [12] = 'Value',
    [13] = 'Enum',
    [14] = 'Keyword',
    [15] = 'Snippet',
    [16] = 'Color',
    [17] = 'File',
    [18] = 'Reference',
    [19] = 'Folder',
    [20] = 'EnumMember',
    [21] = 'Constant',
    [22] = 'Struct',
    [23] = 'Event',
    [24] = 'Operator',
    [25] = 'TypeParameter'
}

return M
```

Step 7: Build Configuration

Update `Makefile`:

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99 -O2
LIBS = -ljson-c

PLUGIN_NAME = clsp
BACKEND_BIN = $(PLUGIN_NAME)_server

# Installation paths
NVIM_DIR = ~/.local/share/nvim/site/pack/myplugins/start/$(PLUGIN_NAME)
BUILD_DIR = build

.PHONY: all clean install

all: $(BUILD_DIR)/$(BACKEND_BIN)

$(BUILD_DIR):
    mkdir -p $(BUILD_DIR)

$(BUILD_DIR)/$(BACKEND_BIN): src/lsp_server.c src/language_analyzer.c | $(BUILD_DIR)
    $(CC) $(CFLAGS) -o $@ $< src/language_analyzer.c $(LIBS)

clean:
```

```
    rm -rf $(BUILD_DIR)

install: $(BUILD_DIR)/$(BACKEND_BIN)
    cp -r . $(NVIM_DIR)
    cp $(BUILD_DIR)/$(BACKEND_BIN) $(NVIM_DIR)/

uninstall:
    rm -rf $(NVIM_DIR)
```

Step 8: Vim Plugin File

Create `plugin/clsp.vim`:

```
if exists('g:loaded_clsp')
  finish
endif
let g:loaded_clsp = 1

" Setup command
command! CLSPSetup lua require('clsp').setup()

" Key mappings (these will be set up in init.lua)
" gD - Go to definition
" K - Hover
" <leader>ca - Code action

" Auto-commands
augroup CLSP
  autocmd!
  autocmd FileType c,cpp,custom_lang lua require('clsp').setup()
augroup END
```

Usage Example

To use this C-based LSP plugin:

1. **Build and install:**

```
make
make install
```

2. **Configure in your Neovim:**

```
-- In your init.lua
require('clsp').setup()

-- Or integrate with built-in LSP
vim.lsp.start({
    name = 'c-lsp-server',
```

```
    cmd = { vim.fn.stdpath('data') ..
'/site/pack/myplugins/start/clsp/build/clsp_server' },
    on_attach = require('clsp').on_attach,
})
```

This LSP plugin approach allows you to create a fully-featured language server in C that can provide:

- Syntax analysis and highlighting
- Code completion
- Go-to-definition
- Hover information
- Code actions
- Document symbols
- And more advanced language features

The C backend provides performance benefits for complex analysis tasks while maintaining full compatibility with Neovim's LSP client infrastructure.

Advanced C-Backend Plugin: Real-time Code Analysis and Visualization Suite

Let me inspire you with a **revolutionary approach** - a C-based plugin that creates **real-time code visualization and analysis tools** that go beyond traditional LSP functionality. This plugin will provide live visual feedback, performance monitoring, and interactive code exploration.

The Vision: Interactive Code Visualization Suite

```
vizcode-nvim/
├── plugin/
│   └── vizcode.vim
├── lua/
│   └── vizcode/
│       ├── init.lua
│       ├── visualizer.lua
│       ├── profiler.lua
│       ├── network.lua
│       └── ui.lua
├── src/
│   ├── viz_server.c          # Main visualization server
│   ├── code_analyzer.c       # Real-time code analysis
│   ├── perf_monitor.c        # Performance monitoring
│   ├── graph_engine.c        # Graph visualization engine
│   └── web_renderer.c        # Web-based visualization
├── assets/
│   ├── viz.html              # Web visualization interface
│   ├── style.css
│   └── viz.js
├── Makefile
└── README.md
```

Step 1: Real-time Code Analysis Engine (C Backend)

Create `src/code_analyzer.c`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <json-c/json.h>
#include <sys/stat.h>

// Code analysis data structures
typedef struct {
    char* filename;
    int line_count;
    int function_count;
    int variable_count;
    double complexity_score;
    char** functions;
    int function_capacity;
    int function_count;
} file_analysis_t;

typedef struct {
    char* symbol;
    int line;
    int column;
    int references;
    int type; // 0=function, 1=variable, 2=class, etc.
} symbol_info_t;

typedef struct {
    symbol_info_t* symbols;
    int symbol_count;
    int symbol_capacity;
} symbol_table_t;

// Global analysis state
file_analysis_t current_analysis = {0};
symbol_table_t symbol_table = {0};
pthread_mutex_t analysis_mutex = PTHREAD_MUTEX_INITIALIZER;

// Analyze C-like code
file_analysis_t analyze_file(const char* filename, const char* content) {
    file_analysis_t analysis = {0};
    analysis.filename = strdup(filename);

    // Count lines
    const char* ptr = content;
    while (*ptr) {
```

```
        if (*ptr == '\n') analysis.line_count++;
        ptr++;
    }

    // Analyze for functions, variables, complexity
    ptr = content;
    int brace_depth = 0;
    int paren_depth = 0;

    while (*ptr) {
        // Detect function definitions
        if (isalpha(*ptr) || *ptr == '_') {
            const char* word_start = ptr;
            while (isalnum(*ptr) || *ptr == '_') ptr++;
            int word_len = ptr - word_start;

            // Check if this looks like a function declaration
            if (ptr[0] == '(' && brace_depth == 0) {
                analysis.function_count++;
                if (!analysis.functions) {
                    analysis.functions = malloc(100 * sizeof(char*));
                    analysis.function_capacity = 100;
                }
                if (analysis.function_count < analysis.function_capacity) {
                    analysis.functions[analysis.function_count - 1] =
strndup(word_start, word_len);
                }
            }
        }

        // Track complexity
        if (*ptr == '{') brace_depth++;
        else if (*ptr == '}') brace_depth--;
        else if (*ptr == '(') paren_depth++;
        else if (*ptr == ')') paren_depth--;

        // Count conditionals and loops for complexity
        if (strncmp(ptr, "if", 2) == 0 && !isalnum(ptr[2])) {
            analysis.complexity_score += 1.0;
        } else if (strncmp(ptr, "for", 3) == 0 && !isalnum(ptr[3])) {
            analysis.complexity_score += 1.5;
        } else if (strncmp(ptr, "while", 5) == 0 && !isalnum(ptr[5])) {
            analysis.complexity_score += 1.5;
        } else if (strncmp(ptr, "switch", 6) == 0 && !isalnum(ptr[6])) {
            analysis.complexity_score += 2.0;
        }

        ptr++;
    }

    return analysis;
}
```

```
// Generate analysis report
json_object* generate_analysis_report(file_analysis_t* analysis) {
    json_object* report = json_object_new_object();

    json_object_object_add(report, "filename", json_object_new_string(analysis-
>filename));
    json_object_object_add(report, "lines", json_object_new_int(analysis-
>line_count));
    json_object_object_add(report, "functions", json_object_new_int(analysis-
>function_count));
    json_object_object_add(report, "complexity", json_object_new_double(analysis-
>complexity_score));

    // Function list
    json_object* functions = json_object_new_array();
    for (int i = 0; i < analysis->function_count && i < analysis->function_capacity;
i++) {
        if (analysis->functions[i]) {
            json_object_array_add(functions, json_object_new_string(analysis-
>functions[i]));
        }
    }
    json_object_object_add(report, "function_list", functions);

    return report;
}

// Thread-safe analysis update
void update_analysis(const char* filename, const char* content) {
    pthread_mutex_lock(&analysis_mutex);

    // Free old analysis
    if (current_analysis.filename) free(current_analysis.filename);
    if (current_analysis.functions) {
        for (int i = 0; i < current_analysis.function_count; i++) {
            if (current_analysis.functions[i]) free(current_analysis.functions[i]);
        }
        free(current_analysis.functions);
    }

    // Perform new analysis
    current_analysis = analyze_file(filename, content);

    pthread_mutex_unlock(&analysis_mutex);
}
```

Step 2: Performance Monitoring Backend

Create `src/perf_monitor.c`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <pthread.h>
#include <json-c/json.h>

// Performance metrics
typedef struct {
    double cpu_usage;
    long memory_usage;
    int file_size;
    double analysis_time;
    time_t last_update;
} perf_metrics_t;

perf_metrics_t current_metrics = {0};
pthread_mutex_t perf_mutex = PTHREAD_MUTEX_INITIALIZER;

// Monitor system resources
void monitor_performance() {
    pthread_mutex_lock(&perf_mutex);

    // Get current time
    current_metrics.last_update = time(NULL);

    // In a real implementation, you'd get actual system metrics
    // This is a simulation
    current_metrics.cpu_usage = (rand() % 1000) / 10.0; // 0.0 to 100.0
    current_metrics.memory_usage = rand() % 1000000; // Random memory usage
    current_metrics.analysis_time = (double)(rand() % 1000) / 1000.0; // 0.0 to 1.0
seconds

    pthread_mutex_unlock(&perf_mutex);
}

// Generate performance report
json_object* generate_perf_report() {
    pthread_mutex_lock(&perf_mutex);

    json_object* report = json_object_new_object();
    json_object_object_add(report, "cpu_usage",
json_object_new_double(current_metrics.cpu_usage));
    json_object_object_add(report, "memory_usage",
json_object_new_int64(current_metrics.memory_usage));
    json_object_object_add(report, "analysis_time",
json_object_new_double(current_metrics.analysis_time));
    json_object_object_add(report, "timestamp",
json_object_new_int64(current_metrics.last_update));

    pthread_mutex_unlock(&perf_mutex);
```

```
    return report;
}
```

Step 3: Main Visualization Server

Create `src/viz_server.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <json-c/json.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "code_analyzer.c"
#include "perf_monitor.c"

#define PORT 8080
#define MAX_CLIENTS 10

typedef struct {
    int socket;
    pthread_t thread;
} client_t;

client_t clients[MAX_CLIENTS];
int client_count = 0;
pthread_mutex_t client_mutex = PTHREAD_MUTEX_INITIALIZER;

// WebSocket message handler
void handle_websocket_message(const char* message, int client_socket) {
    json_object* request = json_tokener_parse(message);
    if (!request) return;

    json_object* type_obj;
    if (!json_object_object_get_ex(request, "type", &type_obj)) {
        json_object_put(request);
        return;
    }

    const char* type = json_object_get_string(type_obj);

    if (strcmp(type, "analyze") == 0) {
        json_object* filename_obj, *content_obj;
        if (json_object_object_get_ex(request, "filename", &filename_obj) &&
            json_object_object_get_ex(request, "content", &content_obj)) {
```

```
            const char* filename = json_object_get_string(filename_obj);
            const char* content = json_object_get_string(content_obj);

            update_analysis(filename, content);

            pthread_mutex_lock(&analysis_mutex);
            json_object* report = generate_analysis_report(&current_analysis);
            json_object_object_add(report, "type",
json_object_new_string("analysis_update"));

            const char* json_str = json_object_to_json_string(report);
            send(client_socket, json_str, strlen(json_str), 0);

            json_object_put(report);
            pthread_mutex_unlock(&analysis_mutex);
        }
    }
    else if (strcmp(type, "perf") == 0) {
        json_object* perf_report = generate_perf_report();
        json_object_object_add(perf_report, "type",
json_object_new_string("performance_update"));

        const char* json_str = json_object_to_json_string(perf_report);
        send(client_socket, json_str, strlen(json_str), 0);

        json_object_put(perf_report);
    }

    json_object_put(request);
}

// Client handling thread
void* handle_client(void* arg) {
    int client_socket = *(int*)arg;
    char buffer[8192];

    while (1) {
        int bytes_read = recv(client_socket, buffer, sizeof(buffer) - 1, 0);
        if (bytes_read <= 0) break;

        buffer[bytes_read] = '\0';
        handle_websocket_message(buffer, client_socket);
    }

    close(client_socket);
    return NULL;
}

// Start the visualization server
void start_viz_server() {
    int server_fd, new_socket;
    struct sockaddr_in address;
```

```
    int opt = 1;
    int addrlen = sizeof(address);

    // Create socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Set socket options
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))) {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    printf("Visualization server started on port %d\n", PORT);

    while (1) {
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen)) < 0) {
            perror("accept");
            exit(EXIT_FAILURE);
        }

        pthread_mutex_lock(&client_mutex);
        if (client_count < MAX_CLIENTS) {
            clients[client_count].socket = new_socket;
            pthread_create(&clients[client_count].thread, NULL, handle_client,
&new_socket);
            client_count++;
        } else {
            close(new_socket);
        }
        pthread_mutex_unlock(&client_mutex);
    }
}

int main() {
```

```
    printf("Starting Code Visualization Server...\n");
    start_viz_server();
    return 0;
}
```

Step 4: Lua Frontend for Visualization

Create `lua/vizcode/visualizer.lua`:

```
local M = {}

local Job = require('plenary.job')
local http = require('plenary.http')

local viz_server_process = nil
local viz_port = 8080

function M.start_visualization_server()
    if viz_server_process then
        print('Visualization server already running')
        return
    end

    local server_path = vim.fn.fnamemodify(debug.getinfo(1).source:match('@?(.*)'),
':p:h:h') .. '/build/viz_server'

    viz_server_process = Job:new({
        command = server_path,
        on_stderr = function(_, data)
            if data then
                print('Viz Server: ' .. data)
            end
        end,
        on_exit = function()
            print('Visualization server stopped')
            viz_server_process = nil
        end,
    })

    viz_server_process:start()

    -- Wait a bit for server to start
    vim.defer_fn(function()
        print('Visualization server started on port ' .. viz_port)
        M.open_visualization_window()
    end, 1000)
end

function M.open_visualization_window()
    -- Open a browser or floating window with visualization
    local url = 'http://localhost:' .. viz_port
```

```
    -- Try to open in browser
    local success = vim.fn.system('xdg-open "' .. url .. '" 2>/dev/null &')
    if vim.v.shell_error ~= 0 then
        -- Fallback: show URL in a message
        vim.notify('Open visualization at: ' .. url, vim.log.levels.INFO)
    end
end

function M.send_code_to_visualizer(filename, content)
    if not viz_server_process then
        print('Visualization server not running')
        return
    end

    local data = {
        type = 'analyze',
        filename = filename,
        content = content
    }

    -- Send via HTTP or WebSocket (simplified)
    http.post('http://localhost:' .. viz_port, {
        body = vim.json.encode(data),
        headers = { ['Content-Type'] = 'application/json' }
    })
end

function M.update_visualization()
    local bufnr = vim.api.nvim_get_current_buf()
    local filename = vim.api.nvim_buf_get_name(bufnr)
    local content = table.concat(vim.api.nvim_buf_get_lines(bufnr, 0, -1, false),
'\n')

    M.send_code_to_visualizer(filename, content)
end

return M
```

Step 5: Advanced UI Module

Create `lua/vizcode/ui.lua`:

```
local M = {}

local api = vim.api
local ns_id = api.nvim_create_namespace('vizcode')

function M.create_complexity_highlights(bufnr)
    local lines = api.nvim_buf_get_lines(bufnr, 0, -1, false)
```

```
    -- Clear existing highlights
    api.nvim_buf_clear_namespace(bufnr, ns_id, 0, -1)

    for i, line in ipairs(lines) do
        -- Analyze line complexity and highlight accordingly
        local complexity = M.calculate_line_complexity(line)

        if complexity > 2 then
            -- High complexity - red highlight
            api.nvim_buf_add_highlight(bufnr, ns_id, 'Error', i - 1, 0, -1)
        elseif complexity > 1 then
            -- Medium complexity - yellow highlight
            api.nvim_buf_add_highlight(bufnr, ns_id, 'WarningMsg', i - 1, 0, -1)
        end
    end
end

function M.calculate_line_complexity(line)
    local complexity = 0

    -- Count nested structures
    local nesting = 0
    for char in line:gmatch('.') do
        if char == '(' or char == '[' or char == '{' then
            nesting = nesting + 1
            complexity = complexity + 0.5
        elseif char == ')' or char == ']' or char == '}' then
            nesting = nesting - 1
        end
    end

    -- Count operators
    for op in line:gmatch('[%+%-%*/<>=!&|]') do
        complexity = complexity + 0.2
    end

    -- Count function calls
    complexity = complexity + select(2, line:gsub('%b()', '')) * 0.3

    return complexity
end

function M.show_function_tree(bufnr)
    local lines = api.nvim_buf_get_lines(bufnr, 0, -1, false)
    local functions = {}

    for i, line in ipairs(lines) do
        -- Simple function detection (improve with regex or parser)
        local func_match = line:match('^%s*(%w+)%s*%(')
        if func_match then
            table.insert(functions, {
                name = func_match,
```

```
                line = i,
                signature = line
            })
        end
    end

    -- Create a floating window with function tree
    local win, buf = M.create_floating_window()

    local content = {}
    table.insert(content, '# Function Tree')
    for _, func in ipairs(functions) do
        table.insert(content, string.format('- %s (line %d)', func.name, func.line))
    end

    api.nvim_buf_set_lines(buf, 0, -1, false, content)

    return win, buf
end

function M.create_floating_window()
    local width = math.floor(vim.o.columns * 0.6)
    local height = math.floor(vim.o.lines * 0.6)

    local buf = api.nvim_create_buf(false, true)

    local win = api.nvim_open_win(buf, true, {
        relative = 'editor',
        width = width,
        height = height,
        row = math.floor((vim.o.lines - height) / 2),
        col = math.floor((vim.o.columns - width) / 2),
        style = 'minimal',
        border = 'rounded'
    })

    return win, buf
end

return M
```

Step 6: Profiling and Network Analysis

Create `lua/vizcode/profiler.lua`:

```
local M = {}

local profile_data = {}
local profiling_enabled = false

function M.start_profiling()
```

```lua
    profiling_enabled = true
    profile_data = {
        start_time = os.time(),
        functions = {},
        memory_usage = {},
        performance_metrics = {}
    }

    -- Start periodic sampling
    M.start_sampling()
end

function M.stop_profiling()
    profiling_enabled = false

    -- Generate report
    local report = M.generate_profile_report()
    M.display_profile_report(report)
end

function M.start_sampling()
    if not profiling_enabled then return end

    -- Sample every 100ms
    vim.defer_fn(function()
        if profiling_enabled then
            M.sample_current_state()
            M.start_sampling() -- Recursive call for continuous sampling
        end
    end, 100)
end

function M.sample_current_state()
    local bufnr = vim.api.nvim_get_current_buf()
    local filename = vim.api.nvim_buf_get_name(bufnr)
    local lines = vim.api.nvim_buf_get_lines(bufnr, 0, -1, false)

    -- Collect metrics
    local metrics = {
        timestamp = os.time(),
        line_count = #lines,
        char_count = 0,
        complexity = 0
    }

    for _, line in ipairs(lines) do
        metrics.char_count = metrics.char_count + #line
        metrics.complexity = metrics.complexity + M.calculate_line_complexity(line)
    end

    table.insert(profile_data.performance_metrics, metrics)
end
```

```lua
function M.calculate_line_complexity(line)
    -- Implementation from UI module
    local complexity = 0
    local nesting = 0

    for char in line:gmatch('.') do
        if char == '(' or char == '[' or char == '{' then
            nesting = nesting + 1
            complexity = complexity + 0.5
        elseif char == ')' or char == ']' or char == '}' then
            nesting = nesting - 1
        end
    end

    complexity = complexity + select(2, line:gsub('[%+%-%*/<>=!&|]', '')) * 0.2
    return complexity
end

function M.generate_profile_report()
    local total_time = os.time() - profile_data.start_time
    local metrics = profile_data.performance_metrics

    if #metrics == 0 then
        return { error = 'No data collected' }
    end

    local avg_complexity = 0
    local max_complexity = 0

    for _, metric in ipairs(metrics) do
        avg_complexity = avg_complexity + metric.complexity
        if metric.complexity > max_complexity then
            max_complexity = metric.complexity
        end
    end

    avg_complexity = avg_complexity / #metrics

    return {
        total_time = total_time,
        samples = #metrics,
        avg_complexity = avg_complexity,
        max_complexity = max_complexity,
        metrics = metrics
    }
end

function M.display_profile_report(report)
    if report.error then
        vim.notify('Profiling error: ' .. report.error, vim.log.levels.ERROR)
        return
```

```
    end

    local content = {
        '=== Code Profiling Report ===',
        'Total time: ' .. report.total_time .. ' seconds',
        'Samples collected: ' .. report.samples,
        'Average complexity: ' .. string.format('%.2f', report.avg_complexity),
        'Max complexity: ' .. string.format('%.2f', report.max_complexity),
        '',
        'Detailed metrics:'
    }

    for i, metric in ipairs(report.metrics) do
        if i <= 10 then -- Show first 10 samples
            table.insert(content, string.format(
                '  Sample %d: %d lines, complexity %.2f',
                i, metric.line_count, metric.complexity
            ))
        end
    end

    local win, buf = require('vizcode.ui').create_floating_window()
    vim.api.nvim_buf_set_lines(buf, 0, -1, false, content)
end

return M
```

Step 7: Main Plugin with Advanced Features

Create `lua/vizcode/init.lua`:

```
local M = {}

local visualizer = require('vizcode.visualizer')
local profiler = require('vizcode.profiler')
local ui = require('vizcode.ui')

function M.setup(opts)
    opts = opts or {}

    -- Start visualization server if enabled
    if opts.enable_visualization ~= false then
        visualizer.start_visualization_server()
    end

    -- Set up autocommands for real-time analysis
    vim.api.nvim_create_autocmd({'TextChanged', 'TextChangedI'}, {
        callback = function()
            if opts.enable_realtime_visualization then
                visualizer.update_visualization()
                ui.create_complexity_highlights(0)
```

```
            end
        end,
        desc = 'Real-time code analysis'
    })

    -- Set up key mappings
    vim.api.nvim_set_keymap('n', '<leader>vv',
        ':lua require("vizcode").toggle_visualization()<CR>',
        { noremap = true, silent = true })

    vim.api.nvim_set_keymap('n', '<leader>vp',
        ':lua require("vizcode").start_profiling()<CR>',
        { noremap = true, silent = true })

    vim.api.nvim_set_keymap('n', '<leader>vt',
        ':lua require("vizcode").show_function_tree()<CR>',
        { noremap = true, silent = true })

    vim.api.nvim_set_keymap('n', '<leader>va',
        ':lua require("vizcode").analyze_current_file()<CR>',
        { noremap = true, silent = true })
end

function M.toggle_visualization()
    if vim.g.viz_server_running then
        -- Stop server
        vim.g.viz_server_running = false
        print('Visualization server stopped')
    else
        visualizer.start_visualization_server()
        vim.g.viz_server_running = true
    end
end

function M.start_profiling()
    profiler.start_profiling()
    vim.notify('Code profiling started', vim.log.levels.INFO)
end

function M.stop_profiling()
    profiler.stop_profiling()
    vim.notify('Code profiling stopped', vim.log.levels.INFO)
end

function M.show_function_tree()
    local win, buf = ui.show_function_tree(0)
    vim.api.nvim_buf_set_option(buf, 'filetype', 'markdown')
end

function M.analyze_current_file()
    local bufnr = vim.api.nvim_get_current_buf()
    local filename = vim.api.nvim_buf_get_name(bufnr)
```

```
    local content = table.concat(vim.api.nvim_buf_get_lines(bufnr, 0, -1, false),
'\n')

    -- Send to C backend for analysis
    visualizer.send_code_to_visualizer(filename, content)

    -- Also show local analysis
    M.show_local_analysis(bufnr)
end

function M.show_local_analysis(bufnr)
    bufnr = bufnr or 0
    local lines = vim.api.nvim_buf_get_lines(bufnr, 0, -1, false)

    local stats = {
        lines = #lines,
        chars = 0,
        words = 0,
        complexity = 0,
        functions = 0
    }

    for i, line in ipairs(lines) do
        stats.chars = stats.chars + #line
        stats.words = stats.words + #vim.split(line, '%s+')
        stats.complexity = stats.complexity + ui.calculate_line_complexity(line)

        -- Count functions
        if line:match('^%s*%w+%s*%(') then
            stats.functions = stats.functions + 1
        end
    end

    local content = {
        '=== File Analysis ===',
        'Lines: ' .. stats.lines,
        'Characters: ' .. stats.chars,
        'Words: ' .. stats.words,
        'Functions: ' .. stats.functions,
        'Complexity Score: ' .. string.format('%.2f', stats.complexity)
    }

    local win, buf = ui.create_floating_window()
    vim.api.nvim_buf_set_lines(buf, 0, -1, false, content)
end

-- Advanced feature: Code dependency graph
function M.generate_dependency_graph()
    -- This would use the C backend to analyze dependencies
    -- and generate a graph visualization
    print('Generating dependency graph...')
```

```
    -- In a real implementation, this would call the C backend
    -- to analyze imports/dependencies and create a visual graph
end

return M
```

Step 8: Web-based Visualization Interface

Create `assets/viz.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Code Visualization</title>
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
    <style>
        body { font-family: monospace; margin: 0; padding: 20px; }
        .container { display: grid; grid-template-columns: 1fr 1fr; gap: 20px; }
        .panel { border: 1px solid #ccc; padding: 15px; border-radius: 5px; }
        .metrics { display: flex; justify-content: space-around; }
        .metric { text-align: center; }
        canvas { width: 100% !important; }
    </style>
</head>
<body>
    <h1>Real-time Code Analysis Dashboard</h1>

    <div class="container">
        <div class="panel">
            <h3>File Analysis</h3>
            <div id="file-info"></div>
            <canvas id="complexity-chart"></canvas>
        </div>

        <div class="panel">
            <h3>Performance Metrics</h3>
            <div class="metrics">
                <div class="metric">
                    <div>CPU Usage</div>
                    <div id="cpu-usage">0%</div>
                </div>
                <div class="metric">
                    <div>Memory</div>
                    <div id="memory-usage">0KB</div>
                </div>
                <div class="metric">
                    <div>Analysis Time</div>
                    <div id="analysis-time">0ms</div>
                </div>
            </div>
            <canvas id="perf-chart"></canvas>
```

```
        </div>
    </div>

    <script src="viz.js"></script>
</body>
</html>
```

Step 9: Advanced Build System

Create `Makefile`:

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99 -O2 -pthread
LIBS = -ljson-c -lm

PLUGIN_NAME = vizcode
SERVER_BIN = $(PLUGIN_NAME)_server
ANALYZER_BIN = $(PLUGIN_NAME)_analyzer

NVIM_DIR = ~/.local/share/nvim/site/pack/myplugins/start/$(PLUGIN_NAME)
BUILD_DIR = build

.PHONY: all clean install web

all: $(BUILD_DIR)/$(SERVER_BIN)

$(BUILD_DIR):
    mkdir -p $(BUILD_DIR)

$(BUILD_DIR)/$(SERVER_BIN): src/viz_server.c | $(BUILD_DIR)
    $(CC) $(CFLAGS) -o $@ $< $(LIBS)

web: assets/viz.html assets/viz.js assets/style.css
    mkdir -p $(NVIM_DIR)/assets
    cp assets/* $(NVIM_DIR)/assets/

clean:
    rm -rf $(BUILD_DIR)

install: all web
    mkdir -p $(NVIM_DIR)/{plugin,lua,vizcode/src}
    cp -r plugin lua src Makefile README.md $(NVIM_DIR)/
    cp $(BUILD_DIR)/$(SERVER_BIN) $(NVIM_DIR)/

uninstall:
    rm -rf $(NVIM_DIR)

# Development targets
dev: all
    @echo "Building development version..."
    @$(CC) $(CFLAGS) -DDEBUG -o $(BUILD_DIR)/$(SERVER_BIN)_debug src/viz_server.c
```

```
$(LIBS)

.PHONY: benchmark
benchmark:
    @echo "Running performance benchmarks..."
    @time $(BUILD_DIR)/$(SERVER_BIN) &
    @sleep 2
    @echo "Benchmark complete"
```

Revolutionary Features This Plugin Provides:

1. Real-time Code Visualization

- Live complexity analysis with visual indicators
- Interactive web-based dashboard
- Real-time performance monitoring

2. Advanced Profiling

- CPU and memory usage tracking
- Code complexity metrics
- Performance bottlenecks identification

3. Dependency Analysis

- Function call graphs
- Code structure visualization
- Dependency mapping

4. Interactive Features

- Floating windows with analysis
- Real-time syntax highlighting based on complexity
- Function tree navigation

5. Extensible Architecture

- C backend for performance-critical operations
- Lua frontend for Neovim integration
- Web interface for rich visualization

This approach transforms Neovim into a **full-featured IDE** with real-time code analysis, performance monitoring, and interactive visualization - all powered by a high-performance C backend that can handle complex analysis tasks that would be too slow in pure Lua!
