

# with-chatgpt

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

---

Here's a clear and practical guide to building a **custom Neovim plugin where the main backend logic is written in C**, while Lua acts as the thin integration layer.

---

## 1. Understand the Architecture You're Building

Neovim plugins typically run through **Lua**, but you can delegate heavy work to **native C code** in two ways:

### A. FFI-style shared library (fast, simple)

- You compile your C backend to a **shared library** ( `.so` , `.dll` , `.dylib` ).
- Lua calls your functions using:
  - `vim.loop` (libuv subprocess or async)
  - or `ffi` (from `luajit` -based Neovim)
  - or a thin C wrapper via `luaopen_*` module.

### B. External process over RPC (like LSP servers)

- Your C program runs as a standalone executable.
- Communicates with Neovim via RPC/JSON messages.
- More complex, but perfect for long-running or parallel computations.

For your first plugin: **use shared library → load from Lua → call C functions directly.**

---

## 2. Minimal Project Structure

```
my_nvim_plugin/
├── lua/
│   └── myplugin/
│       └── init.lua
├── csrc/
│   ├── mybackend.c
│   └── mybackend.h
├── Makefile
└── README.md
```

## 3. Write the C Backend (Example)

**csrc/mybackend.c** Simple example: a function that processes a string.

```
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

char* to_uppercase(const char* input) {
    size_t len = strlen(input);
    char* out = malloc(len + 1);
    for (size_t i = 0; i < len; i++) {
        out[i] = toupper(input[i]);
    }
    out[len] = '\0';
    return out;
}
```

Expose this as a **Lua module**:

```
#include <lua.h>
#include <lauxlib.h>

extern char* to_uppercase(const char* input);

static int l_to_upper(lua_State* L) {
    const char* input = luaL_checkstring(L, 1);
    char* result = to_uppercase(input);
    lua_pushstring(L, result);
    free(result);
    return 1;
}

int luaopen_mybackend(lua_State* L) {
    luaL_Reg funcs[] = {
        {"to_upper", l_to_upper},
        {NULL, NULL}
    };
    luaL_newlib(L, funcs);
    return 1;
}
```

You now have a proper **Lua C module**.

---

## 4. Build it into a Shared Library

### Makefile

```
CC=gcc
CFLAGS=-Wall -O2 -fPIC
LDFLAGS=-shared
```

```
TARGET=mybackend.so
SRC=csrc/mybackend.c
```

```
all:
    $(CC) $(CFLAGS) -o $(TARGET) $(SRC) $(LDFLAGS)
```

```
clean:
    rm -f *.so
```

Run:

```
make
```

You get:

```
mybackend.so
```

Move it to:

```
my_nvim_plugin/lua/myplugin/mybackend.so
```

(Neovim can load C modules from Lua paths.)

---

## 5. Lua Layer: Minimal Neovim Plugin

lua/myplugin/init.lua

```
local backend = require("myplugin.mybackend")

local M = {}

function M.uppercase_current_word()
    local word = vim.fn.expand("<cword>")
    local result = backend.to_upper(word)
    print("Uppercase:", result)
end

return M
```

## 6. Provide a Neovim Command or Keymap

Inside `init.lua`, add a command:

```
vim.api.nvim_create_user_command(
    "UppercaseWord",
    function() require("myplugin").uppercase_current_word() end,
```

```
}  
)
```

Now use it inside Neovim:

```
:UppercaseWord
```

Your C backend function will process the current word.

---

## 7. Optional: Async Execution (libuv)

If your C backend needs to run heavy tasks:

- compile it as a standalone CLI tool ( `mybackend_bin` )
- run via `vim.loop.spawn` :

```
vim.loop.spawn("mybackend_bin", {  
  args = {"--task", "xyz"}  
}, function() print("done!") end)
```

This avoids blocking the UI.

---

## 8. Scaling Up Your Plugin

Add features incrementally:

### ✓ Syntax analysis

C backend parses file → Lua displays results.

### ✓ High-performance compute

Image processing, simulation, diff calculation.

### ✓ Tree parsing or AST

Integrate with C parsers (ANTLR, tree-sitter style functions, custom lexers).

### ✓ LSP alternative

Use your C backend as a lightweight protocol server.

---

## 9. Recommended Enhancements

- Add unit tests for your C code (CMocka, Unity).
- Add Lua tests using `plenary.nvim` .
- Add async wrappers through `vim.loop.new_thread` .
- Export more C functions as Lua modules.
- Optionally expose the entire C library with FFI.

---

## 10. If You Want, I Can Provide

Just tell me which direction you want next:

### A. Minimal template project (fully ready)

Directory structure, build script, plugin loader.

### B. More advanced C backend example

Like: tokenizer, markdown parser, diff engine, etc.

### C. Use RPC instead of Lua C API

Example: fast background analyzer.

### D. Generate a full plugin skeleton

With docs, tests, installer, and CI.

---

If you tell me your target purpose (text processing? file scanning? code analysis?), I'll tailor the plugin architecture and code.

---

If your main goal is **expanding or enriching Neovim's API itself**, then a C-powered plugin is a *perfect* choice. You essentially create **new capabilities** that Lua alone cannot deliver—high-performance operations, complex data structures, system-level features, or entirely new interfaces that feel “native” inside Neovim.

Below is a practical roadmap + techniques that let your C backend behave like a **new API layer** for Neovim.

---

## 1. Decide the Type of API Enrichment

There are three powerful patterns:

### A. Extend Neovim's Lua API (fastest & simplest)

You write a **Lua C module** that adds new functions, then expose them into the `vim.api` space.

Example: Add functions like:

- `vim.api.get_words()`
- `vim.api.file_hash(path)`
- `vim.api.tree_parse(buffer)`

This makes your plugin “blend into” Neovim's core API.

---

### B. Expose a full “subsystem” backed by C

Examples:

- Custom buffer operations (fast diff, fast search)
- Custom window rendering helpers

- Custom code analysis “engine”
- Custom virtual text layout engine

This feels like **adding built-in Neovim modules**.

---

## C. Provide new async/parallel API features

Your C backend runs separate threads or processes, then pushes results to Lua callbacks.

Examples:

- `vim.api.async_job_spawn(...)`
- `vim.api.fs_scan(...)` (fast recursive directory scanning)
- `vim.api.lang_query(...)`

This extends Neovim with new performance-level primitives.

---

## 2. Architecture That Makes It Feel Like a “New API”

This approach works extremely well:

```
+-----+
|      Neovim Lua      |
+-----+
|      require(...)    |
|      v               |
+-----+
| Lua wrapper (thin layer) |
+-----+
|      calls C functions  |
|      v               |
+-----+
|      C backend library  |
|      (your “new API” engine) |
+-----+
```

Your C backend becomes the **core engine**, Lua just exposes friendly API calls.

---

## 3. Example: Build New APIs in Neovim

### Goal

Create a module `vim.api.ext` with functions:

```
vim.api.ext.get_line_hash(bufnr, lnum)
vim.api.ext.scan_dir(path)
vim.api.ext.ast_parse(bufnr)
```

These will be powered by your C backend.

---

## 4. C Backend: Provide API Functions

Example C file adding new “API” capabilities:

```
#include <lua.h>
#include <lauxlib.h>
#include <stdint.h>
#include <stdio.h>
#include <dirent.h>
#include <string.h>
#include <openssl/sha.h>

// --- Example 1: Fast hash of a string ---
static int l_hash_string(lua_State* L) {
    size_t len;
    const char* text = luaL_checklstring(L, 1, &len);

    unsigned char digest[SHA256_DIGEST_LENGTH];
    SHA256((unsigned char*)text, len, digest);

    lua_pushlstring(L, (char*)digest, SHA256_DIGEST_LENGTH);
    return 1;
}

// --- Example 2: Directory scanning ---
static int l_scan_dir(lua_State* L) {
    const char* path = luaL_checkstring(L, 1);
    DIR* dir = opendir(path);

    lua_newtable(L);
    int idx = 1;

    if (dir) {
        struct dirent* entry;
        while ((entry = readdir(dir)) != NULL) {
            lua_pushinteger(L, idx++);
            lua_pushstring(L, entry->d_name);
            lua_settable(L, -3);
        }
        closedir(dir);
    }
    return 1;
}

int luaopen_nvimext(lua_State* L) {
    luaL_Reg funcs[] = {
        {"hash_string", l_hash_string},
        {"scan_dir", l_scan_dir},
    }
```

```

        {NULL, NULL}
    };
    luaL_newlib(L, funcs);
    return 1;
}

```

You compile this as a shared library ( `nvimext.so` ).

## 5. Lua Layer: Expose the API as `vim.api.ext`

```

local native = require("nvimext")

vim.api.ext = {
    hash_string = function(str)
        return native.hash_string(str)
    end,

    scan_dir = function(path)
        return native.scan_dir(path)
    end,
}

```

Now inside Neovim:

```

:lua print(#vim.api.ext.hash_string("hello"))
:lua vim.print(vim.api.ext.scan_dir("."))

```

You have successfully **added new APIs to Neovim**.

## 6. Pattern for Large-Scale API Extensions

You can build modules like:

### 1. `vim.api.fs.*`

File hashing, recursive directory scanning, file change monitoring.

### 2. `vim.api.syntax.*`

Tokenizers, lexers, parsers, semantic analysis.

### 3. `vim.api.buffer.*`

Fast:

- diffing
- substring search
- multi-line replace These outperform Lua significantly.



#### 4. `vim.api.jobs.*`

A C-powered job/async system with threads or worker pools.

#### 5. `vim.api.math.*`

Vector/matrix operations, geometry, spatial indexing, etc.

---

## 7. If You Want to Go Deeper: True Neovim Core Integration

Neovim itself is written in C. You can build a plugin that:

- creates new RPC methods
- registers new “native-like” API functions
- integrates with `msgpack-rpc`
- provides your own low-level UI events

This allows something like:

```
nvim_call_function("ExtComputeFast", ["abc"])
```

or even:

```
:ExtSearch hello
```

behaving like a builtin feature.

If you want, I can also guide you to:

✓ how to expose msgpack-RPC functions from C ✓ how to register your module as a full RPC provider ✓ how to hook Neovim events from C ✓ how to build a “native” subsystem inside Neovim

---

## 8. Tell Me the Direction You Want Next

Choose one:

1. **Provide full plugin skeleton** (C + Lua + Makefile)
2. **Show how to add tens of new Neovim API functions**
3. **Show how to expose asynchronous C APIs**
4. **Show how to build an RPC-based C plugin (like treesitter / LSP)**
5. **Make a real-world example (filesystem, syntax engine, diff engine, etc.)**

Which path should we explore?

---

Here are **several new categories of Neovim plugins** that still use **C as the main backend**, but with different behaviors, architectures, and goals. Each type is distinct and unlocks different capabilities inside Neovim—some feel like system extensions, some like language engines, some like UI helpers.

---

# 1. High-Performance Text Engine Plugin

Your C backend becomes a **fast text-processing engine**.

## Examples

- ultra-fast search engine (like ripgrep embedded inside Neovim)
- semantic grep/token scanning
- structural search (AST-based search, but lightweight)
- multi-pattern matcher (Aho–Corasick)

## Concept

C handles raw text + algorithmic heavy lifting → Lua shows results.

## Why powerful

You get features *far* beyond Lua performance.

---

# 2. Buffer Virtualization / Custom Buffer Types

C implements a **virtual buffer type**:

## Examples

- virtual buffer backed by an SQLite database
- virtual buffer backed by remote server
- “view-only” buffer rendering external data (e.g., image metadata, binary dump)
- procedurally generated buffer content (logs, memory map, binary viewer)

## What this enables

You can simulate new “file systems” or “views” inside Neovim.

---

# 3. Native UI Extension Plugin

C draws or computes extra UI behavior.

## Examples

- inline image renderer (Kitty graphics protocol, sixel, etc.)
- enhanced renderer for Markdown preview
- custom minimap engine
- layout engine for complex virtual text

## C Backend Roles

- generates pixel/ASCII/graphical data
  - performs layout computation
  - manages caching for performance
-

## 4. Async/System Integration Plugin

Your C backend becomes a **system daemon**, exposing:

### Examples

- Linux inotify/fanotify events → Neovim events
- GPU statistics monitoring
- CPU/memory sensor monitor
- network watcher (websocket or tcp client)

### Lua side

You emit:

```
vim.api.nvim_exec_autocmds("User", {pattern="SystemEvent"})
```

This feels like adding new Neovim autocommands.

---

## 5. Embedded DSL or Script Runtime Plugin

A plugin where the **C backend is an interpreter** for a mini-language.

### Ideas

- embedded scripting language (tiny-lisp, forth, lua subset)
- markdown macro interpreter
- configuration templating engine
- domain-specific compiler (for notes, configs, widgets)

### Outcome

Users write tiny scripts in buffers → C executes them.

---

## 6. Neovim LSP Alternative (micro-LSP)

A compact LSP server written in C *but designed only for Neovim*.

### Examples

- ultralight C tokenizer-based LSP (no full AST)
- specialized LSP for custom file formats
- real-time linting engine using threads

### Why unique

Instead of full LSP spec, you define **your own protocol**, optimized for speed.

---

## 7. Command-Line Tool Integrator Plugin

C backend acts as a wrapper for system utilities.

### Examples

- wrapper for git functionality (libgit2)
- wrapper for sqlite embeddings
- wrapper for imagemagick operations (resize, convert)
- wrapper for ffmpeg metadata parsing

### Neovim gets new commands:

```
:GitHistory  
:DBQuery  
:ImageInfo  
:MediaMetadata
```

---

## 8. Neovim AI/ML Accelerator Plugin

C backend integrates small machine learning functionality (without Python).

### Examples

- tiny ONNX inference (C inference engine)
- Markov chain text generator
- fast embeddings (word2vec subset)
- code suggestion using probabilistic models

### Effect

Neovim gets lightweight AI features, CPU-efficient.

---

## 9. Graphics/Terminal Media Plugin

Plugin that brings **media handling** to Neovim:

### Examples

- inline image viewer (sixel)
- waveform visualization for audio files
- file/directory tree rendered in pseudo-graphics
- ASCII chart/graph generator using C backend

### Use cases

Make Neovim more TUI-capable than usual.

---

## 10. Neovim Data Science Helper Plugin

C backend handles dataset operations.

### Examples

- CSV parser with streaming
- data aggregation engine
- histogram + stats computation
- JSON/CBOR reader with fast parsing

Lua only displays the result beautifully.

---

## 11. Structured Document Engine Plugin

A plugin that parses complex formats entirely via C:

### Examples

- TOML/YAML custom parser
- Markdown parser (like md4c) but customized
- Org-mode parser
- binary file format viewer (ELF, PE, WASM)

### Lua layer

Creates commands like:

```
:DocOutline  
:BinaryDump  
:FormatTree
```

---

## 12. Game or Simulation Plugin

Use C to compute simulation/game logic, visualize in buffer:

### Examples

- cellular automata
- physics mini-simulation
- map grid renderer
- ascii roguelike inside Neovim
- memory visualizer via simulation backend

This is great fun and shows off raw compute.

---

## 13. Workspace Manager / Project Indexer

C backend builds and maintains **incremental indices** of project files.

### Examples

- real-time file indexer (faster than telescope)
- tag generator or ctags-compatible indexer
- dependency graph builder

- code symbol searcher with cache

### Features

Plugins like:

```
:ProjectSymbols  
:FileSearch  
:DependencyTree
```

---

## 14. Platform API Plugin (OS Integration)

Expose low-level system APIs into Neovim:

### Examples

- POSIX system calls (memory info, processes, uptime)
- Windows API bindings
- Linux syscalls
- kernel event streams

Lua gets new API such as:

```
vim.api.sys.get_process_list()  
vim.api.sys.disk_usage()
```

---

## 15. Performance Profiler Plugin

C backend instruments Neovim's runtime:

### Examples

- event tracing
- startup profiler
- buffer operation profiler
- memory allocation tracking

Shows results in dashboard buffer.

---

## If you want, I can build one of these types:

Tell me the number/style you want:

**1. Custom virtual buffer system**

**2. Real-time system event plugin**

**3. High-performance text engine**

**4. Neovim UI extension module**

## 5. Tiny-LSP in C

## 6. System integration API

## 7. Image/graphics plugin

## 8. Indexer / search engine

## 9. Data parser / doc processor

## 10. Your own category (you describe idea)

Pick one and I'll generate a full architecture + template.

---

Here are **even more unconventional, extreme, and highly creative Neovim plugin ideas** where the **C backend** becomes the *core engine*. These go far beyond typical Neovim plugins—they push Neovim into areas it normally cannot reach.

Let's go deeper, weirder, more powerful.

---

# 1. “Kernel-like” Background Service Embedded in Neovim

Your C backend runs a long-lived engine inside Neovim:

### Capabilities

- event dispatcher
- scheduler
- incremental file indexer
- lightweight message bus for plugins

### Effect

Neovim becomes a mini operating system for text workflows.

Lua plugins can subscribe:

```
on("FileIndexed", function(ev) ... end)
```

C backend emits events at high speed.

---

# 2. Neovim as a Hardware Operations Console

Use C to access hardware-level APIs:

### Ideas

- USB device explorer
- GPIO interaction (Raspberry Pi)
- microcontroller programmer/serial monitor

- Bluetooth/WiFi scanning engine

Inside Neovim, you get commands like:

```
:USBList
:FlashMicrocontroller firmware.bin
:SerialMonitor
```

Neovim becomes a **hardware engineering dashboard**.

---

## 3. Embedded Database Engine Inside Neovim

Build a **C-backed custom database engine**:

### Capabilities

- persistent key/value store
- fast project-wide key indexing
- time-travel logs
- fuzzy search cache
- metadata tracker (file types, tags, stats)

### Lua-facing API

```
vim.api.db.put("key", "value")
vim.api.db.search("pattern")
vim.api.db.stats()
```

Neovim becomes a knowledge graph.

---

## 4. Neovim as a High-Performance Visualization Terminal

Use a C backend to render:

- ASCII charts
- 2D heatmaps
- 3D wireframes
- spectrograms
- signal plots

All **live-updated** in buffers or floating windows.

Example: Real-time FFT: C processes audio → Neovim shows waveform.

---

## 5. Real-Time Event Loop Plugin

Your C backend becomes a custom async runtime:



## Features

- multitasking via fibers
- timers
- file watchers
- message channels
- long-running workers

Expose to Lua:

```
vim.loopspawn(function()  
  -- cooperative coroutine managed by your C engine  
end)
```

This is like embedding your own **libuv variant**.

---

## 6. High-Performance Semantic Engine

A C backend maintaining a **real-time AST** of the current file or project. Not a full LSP—faster, custom, incremental.

### C Responsibilities

- track changes in buffer
- update token tree
- maintain symbol table
- emit events (“SymbolChanged”, “StructureChanged”)

Applications:

- live fold regions
- semantic selection
- advanced context UI
- mini-rust-analyzer-like features

All custom, minimal, and blazing fast.

---

## 7. Code Intelligence Engine Beyond LSP

Example features:

- cross-language reference finder
- static analysis tailored for your own needs
- macro expansion simulation
- import/require resolution engine
- project architecture mapper

Everything computed by C in milliseconds.

---

## 8. Virtual Machine or Emulator Inside Neovim

Integrate C-based VM or emulator:

- tiny bytecode VM
- JavaScript engine (Duktape, QuickJS)
- WASM runtime
- CHIP-8 emulator
- Forth interpreter
- Brainfuck VM

You can run code directly from a Neovim buffer:

```
:RunBufferInVM
```

Or visually debug step-by-step.

---

## 9. Custom Embedded Renderer (Experimental)

A C backend that interfaces with terminal protocols for graphics:

### Examples

- Kitty graphics
- Sixel
- iTerm2 inline images
- unicode braille pixel-art renderer

Neovim shows images, diagrams, charts, *inline with text*.

---

## 10. Neovim as a Server (Turning Editor → Platform)

Use C to embed:

- HTTP server
- WebSocket server
- RPC service

Then Neovim becomes a live backend for external tools:

- browser-based dashboard
- mobile remote editor
- live performance monitoring UI

Lua interacts via:

```
vim.api.server.broadcast("event", data)
```

---

## 11. Memory Engineering Tools

C backend reads system memory / process info:

- /proc parsing

- stack trace decoding
- lightweight crash analyzer
- memory map viewer for binaries
- live-disassembler (disassemble code under cursor)

Neovim becomes a mini-IDA or debugging console.

---

## 12. Audio / DSP Engine

Plugin backed by C DSP code:

- real-time audio spectrum
- waveform editing
- beat detection
- MIDI controller integration
- synthesizer embedded in Neovim (why not?)

Your Neovim plugin becomes a **text-based DAW**.

---

## 13. Workflow Automation Engine

A powerful C runtime that:

- observes file system
- runs workflows
- triggers tasks
- schedules builds
- maintains dependency graph

Lua only orchestrates UI and triggers.

You get commands like:

```
:BuildOnSave  
:RunWorkflow test  
:WorkflowGraph
```

---

## 14. Binary Inspection / Reverse Engineering Module

C backend implements binary parsing:

- ELF/PE/Mach-O viewer
- WASM decoder
- section/segment visualizer
- symbol table reader

Lua displays the structured output in fancy tree buffers.

---

## 15. GPU Shader Tester / Compute Module

With OpenGL/Vulkan API calls from C:

- compile shaders from buffer
- run compute shaders
- visualize results via ASCII heatmaps
- profile shader execution time

Great for graphics programmers who live in Vim.

---

## 16. Real-Time Collaboration Engine

Use a C backend to implement a high-performance sync engine:

- CRDT
- OT (Operational Transform)
- network sync
- peer-to-peer discovery

Lua handle UI, but C handles consistency model.

You get Collaborative Neovim *offline-capable*.

---

## If you want, I can expand on:

- Architecture diagrams
- C/Lua API designs
- Build system templates
- Full working example plugin
- Implementation for any idea above

Just pick one idea or ask for **more “extreme” directions**.

---

# with-grok

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 a C Program as the Main Backend

Neovim plugins are typically written in **Lua** (preferred for performance and integration) or Vimscript, but you can use **any language** for heavy computation or performance-critical parts. Since you want a **C program as the main backend**, the best (and officially supported) way is to create a **remote plugin** (also called an rplugin) that runs as a separate process and communicates with Neovim via **MessagePack-RPC**.

This approach:

- Keeps Neovim fast (no heavy C code loaded into the process).
- Allows your C binary to do the real work (e.g., fast processing, native libraries, etc.).
- Exposes Vim commands, functions, or autocommands that call into your C code.

Neovim does **not** have a built-in "C host" like it does for Python or Node.js, so you have to implement the low-level MessagePack-RPC protocol yourself in C. It's doable but requires some boilerplate.

### Step 1: Understand the Architecture

```
Neovim <---> stdio (MessagePack-RPC) <---> Your C executable (remote plugin host)
```

- Neovim discovers your plugin via a small Lua/Vimscript manifest in `rplugin/manifest`.
- When a user calls a command/function defined by your plugin, Neovim starts your C binary (if not running) and talks to it over stdin/stdout using MessagePack-RPC.
- Your C program must:
  - Read RPC requests (type 0) and notifications (type 2).
  - Handle "specs" request at startup to register functions/commands.
  - Implement the actual logic for your functions.

### Step 2: Plugin Directory Structure

Standard pack-style layout (works with any plugin manager):

```
~/.local/share/nvim/site/pack/myplugins/start/my-c-plugin/  
├─ lua/  
│   └─ my_c_plugin.lua          -- optional Lua wrapper (recommended)  
├─ pythonx/ or whatever/      -- not needed  
└─ rplugin/c/  
    └─ my_c_host                -- your compiled C executable (no extension on  
Linux/macOS)
```

Or simpler for pure remote plugins:

```
~/.config/nvim/rplugin/c/my_c_host  -- the binary
```

But the pack layout is better for distribution.

### Step 3: Write the C Backend (the "host")

You need a MessagePack library for C. Recommended:

- **msgpack-c** (<https://github.com/msgpack/msgpack-c>) – very popular and lightweight.

Install it (e.g., via your package manager or build from source).

Here's a **minimal working example** of a C remote plugin host that exposes one function: `Add(a, b)` → returns `a + b`.

```
// my_c_host.c
#include <stdio.h>
#include <stdlib.h>
#include <msgpack.h>
#include <string.h>

typedef struct {
    msgpack_sbuffer sbuf;
    msgpack_packer pk;
} Packer;

void init_packer(Packer *p) {
    msgpack_sbuffer_init(&p->sbuf);
    msgpack_packer_init(&p->pk, &p->sbuf, msgpack_sbuffer_write);
}

void send_response(Packer *p, uint32_t msgid, const msgpack_object *result, const
msgpack_object *error) {
    msgpack_sbuffer tmp;
    msgpack_sbuffer_init(&tmp);
    msgpack_packer tmp_pk;
    msgpack_packer_init(&tmp_pk, &tmp, msgpack_sbuffer_write);

    // Response format: [type=1, msgid, error, result]
    msgpack_pack_array(&tmp_pk, 4);
    msgpack_pack_uint8(&tmp_pk, 1);
    msgpack_pack_uint32(&tmp_pk, msgid);
    if (error) msgpack_pack_object(&tmp_pk, *error);
    else msgpack_pack_nil(&tmp_pk);
    if (result) msgpack_pack_object(&tmp_pk, *result);
    else msgpack_pack_nil(&tmp_pk);

    // Write length prefix + data (Neovim expects this on stdio)
    uint32_t len = htonl((uint32_t)tmp.size);
    fwrite(&len, sizeof(len), 1, stdout);
    fwrite(tmp.data, tmp.size, 1, stdout);
    fflush(stdout);

    msgpack_sbuffer_destroy(&tmp);
}
```

```

int main() {
    msgpack_unpacker unpack;
    msgpack_unpacker_init(&unpack, MSGPACK_UNPACKER_INIT_BUFFER_SIZE);

    // First request is always "specs" (msgid usually 1)
    // We hardcode the response for simplicity (real plugins generate this
    dynamically)
    while (1) {
        // Read length prefix
        uint32_t len;
        if (fread(&len, sizeof(len), 1, stdin) != 1) break;
        len = ntohl(len);

        // Resize unpacker buffer and read payload
        msgpack_unpacker_reserve_buffer(&unpack, len);
        if (fread(msgpack_unpacker_buffer(&unpack), len, 1, stdin) != 1) break;
        msgpack_unpacker_buffer_consumed(&unpack, len);

        msgpack_unpacked und;
        msgpack_unpacked_init(&und);
        if (!msgpack_unpacker_next(&unpack, &und)) continue;

        msgpack_object obj = und.data;
        if (obj.type != MSGPACK_OBJECT_ARRAY || obj.via.array.size != 4) continue;

        uint8_t type = obj.via.array.ptr[0].via.u64;
        uint32_t msgid = obj.via.array.ptr[1].via.u64;
        // ptr[2] = method name (string)
        // ptr[3] = params (array)

        if (type == 0) { // request
            msgpack_object method = obj.via.array.ptr[2];
            msgpack_object params = obj.via.array.ptr[3];

            if (method.via.str.size == 5 && strcmp(method.via.str.ptr, "specs", 5)
== 0) {

                // Respond with plugin spec (register one function: Add)
                msgpack_object_map map;
                // Build the specs map (simplified, see full format in :h remote-
plugin)
                // For brevity, this is a minimal placeholder. In real code, build
proper dict.
                const char *specs = "[{'type': 'function', 'name': 'Add', 'sync':
v:true, 'opts': {}}]";
                // Actually pack proper msgpack here (omitted for brevity)
                // Use a pre-packed binary or build it.
                // Real example repos do this properly.
                msgpack_object nil_obj = {.type = MSGPACK_OBJECT_NIL};
                send_response(NULL, msgid, &nil_obj, NULL); // placeholder
            } else if (/* method == "Add" */) {
                // Parse params: [a, b]

```

```

        int64_t a = params.via.array.ptr[0].via.i64;
        int64_t b = params.via.array.ptr[1].via.i64;
        msgpack_object result;
        result.type = MSGPACK_OBJECT_POSITIVE_INTEGER;
        result.via.u64 = a + b;
        msgpack_object nil_err = {.type = MSGPACK_OBJECT_NIL};
        send_response(NULL, msgid, &result, &nil_err);
    }
}

msgpack_unpacked_destroy(&und);
}

msgpack_unpacker_destroy(&unpk);
return 0;
}

```

#### Compile:

```
gcc -O2 -o my_c_host my_c_host.c -lmsgpackc
```

This example is simplified – full implementations exist (see references below). The tricky part is packing the initial "specs" response correctly.

#### Step 4: Register the Plugin (Manifest)

Neovim needs a manifest. The easiest way is to run `:UpdateRemotePlugins` after placing the binary, but for C you often need a tiny Lua shim to force registration.

Create `lua/my_c_plugin.lua` :

```

vim.api.nvim_command([[runtime! plugin/rplugin.vim]])
vim.remote.plugins.register({
  name = 'MyCPlugin',
  type = 'c', -- not standard, but works if binary is executable
  files = { vim.fn.expand('~/.config/nvim/rplugin/c/my_c_host') },
})

```

Then run `:UpdateRemotePlugins` and restart Neovim.

#### Step 5: Use the Plugin

After registration, you can call:

```
echo rpcrequest(g:channel_id, 'Add', 2, 3) " → 5
```

Or expose a nice Vim command/function in Lua.

#### Real-World Examples & Better Starting Points

- **Full C remote plugin example:** <https://github.com/neovim/neovim/issues/7383#issuecomment-427000531> (old but useful)



- **msgpack-c + Neovim RPC:** Search GitHub for "neovim msgpack c" – there are a few proof-of-concepts.
- Alternative easier path: Write the heavy part as a normal C CLI tool and call it from a Lua plugin using `vim.system` (Neovim 0.10+) or `vim.fn.system`. This avoids RPC entirely and is often sufficient.

```
local function fast_compute_with_c(input)
  local output = vim.system({'/path/to/my_c_tool', input}, {text =
true}):wait().stdout
  return output
end
```

This is **much simpler** and recommended unless you truly need bidirectional RPC (e.g., subscriptions, events from C to Neovim).

If you need the full remote plugin power (autocommands, async, etc.), start from the msgpack-c example and expand the specs response.

Let me know what your plugin is supposed to do (e.g., parsing, computation, GUI?), and I can give a more targeted example! This is advanced territory, but totally doable and very powerful. 🚀

If your goal is to **enrich Neovim’s own APIs** (i.e. add brand-new `vim.api.nvim_*` functions or new global Vimscript/Lua functions that feel like they belong to Neovim itself), then a pure remote plugin in C is possible but **not the most ergonomic way** anymore in 2025.

Here are the modern, powerful, and actually used approaches in 2025, ranked from “feels most native” to “raw power”:

Goal	Best Approach in 2025	Why it’s better	Performance	Example Projects
Add new <code>vim.api.nvim_*</code> functions that feel 100% native	<b>Libnvim C extension</b> (dynamic library loaded into Neovim process)	Functions appear exactly like built-in <code>nvim_*</code> APIs, zero RPC overhead, full access to Neovim C internals	Native C speed	<code>nvim-treesitter</code> , <code>neovim/nvim-lspconfig</code> ’s C parts, <code>nvim-oxi</code>
Add new Lua functions ( <code>vim.myplugin.xxx</code> ) with C speed	<b>nvim-oxi</b> (Rust) or <b>LuaJIT FFI</b> calling your C code	Almost zero overhead, safe Rust or direct FFI, no MessagePack	Near-native	Most new high-perf plugins use <code>oxi</code> now
You love C and want to stay in C, but still want native APIs	<b>Write a libnvim-based shared library</b> ( <code>.so/.dll/.dylib</code> )	Same as row 1, but you write in pure C instead of Rust	Native	Old plugins like <code>vim-eunuch</code> ’s C parts
You just need fast computation, not new APIs	<b>Lua plugin + call C binary</b> with <code>vim.system</code> (sync) or <code>vim.uv.spawn</code> (async)	Zero boilerplate, easy debugging	Very fast (only IPC cost)	<code>nvim-silicon</code> , <code>glow.nvim</code>

You insist on remote plugin in C	MessagePack-RPC host in C	Works, but huge boilerplate and latency	Slowest	Almost nobody does this in 2025
----------------------------------	---------------------------	---	---------	---------------------------------

## Recommended Path in 2025: libnvim + C (if you love C) or nvim-oxi (if you're open to Rust)

### Option 1 – Pure C: Write a real Neovim C plugin (libnvim extension)

This is the **only way** to truly enrich `vim.api` with new C functions.

Neovim exposes its full C API via `libnvim`. You write a shared library that registers new API functions on startup.

Example: Add `nvim_myplugin_superfast_parse(buffer, lang)` that parses 10× faster than Lua.

```
// myplugin.c
#include <lauxlib.h>
#include <lua.h>
#include <lualib.h>
#include <nvim/api/private/helpers.h>
#include <nvim/api/nvim.h>
#include <nvim/generators.h>

// Your fast C parser
String my_superfast_parser(Buffer buffer, String lang) {
    // ... your blazing-fast C code ...
    return (String){.data = xstrdup("parsed result"), .size = strlen("parsed result")};
}

static int lua_myplugin_parse(lua_State *L) {
    Buffer buffer = nvim_check_buffer(L, 1);
    String lang = luaL_checkstring(L, 2); // simplified

    String result = my_superfast_parser(buffer, lang);
    lua_pushstring(L, result.data);
    kvec_push(String, result); // free later if needed
    return 1;
}

// Called automatically when the .so is loaded
__attribute__((constructor)) static void init(void) {
    // Get the Lua state that Neovim uses
    lua_State *L = nvim_lua_state();

    // Expose as vim.myplugin.parse(...)
    lua_getglobal(L, "vim");
    lua_newtable(L);
    lua_pushcfunction(L, lua_myplugin_parse);
    lua_setfield(L, -2, "parse");
}
```

```
lua_setfield(L, -2, "myplugin");
}
```

Compile:

```
gcc -shared -fPIC -o myplugin.so myplugin.c \
-I/usr/include/nvim -lnvim -llua5.1 # paths depend on your system
```

Put it in `~/.config/nvim/plugin/myplugin.so` or package it properly.

Now in Lua you can just do:

```
print(vim.myplugin.parse(0, "python")) -- feels completely native
```

Real-world examples:

- <https://github.com/nvim-treesitter/nvim-treesitter> (C parsers loaded this way)
- <https://github.com/neovim/neovim/tree/master/src/nvim> (see how they add APIs)

### Option 2 – The 2025 mainstream: nvim-oxi (Rust)

Everyone who wants C-level speed but hates C memory bugs uses this now.

```
// Cargo.toml
[package]
name = "myplugin"
edition = "2021"

[lib]
crate-type = ["cdylib"]

[dependencies]
nvim-oxi = "0.5"
```

```
// src/lib.rs
use nvim_oxi::api;

#[api]
pub fn superfast_parse(bufnr: u32, lang: &str) -> String {
    // your blazing-fast Rust (or call C via FFI)
    "parsed result".to_string()
}

#[nvim_oxi::module]
fn init() -> nvim_oxi::Result<()> {
    Ok(())
}
```

Build → `myplugin.so` → drop into Neovim → you instantly have `vim.myplugin.superfast_parse`.

Zero boilerplate, memory safe, async support, etc.

When to still use a C binary (not loaded into process)

If your C code:

- links huge libraries (LLVM, libclang, webkit, etc.)
- does very long-running work
- needs to be sandboxed

Then just call it from Lua:

```
-- init.lua
vim.api.nvim_create_user_command("MyFastGrep", function(opts)
  local output = vim.system({'/path/to/my_c_grep', opts.args}, {text = true}):wait()
  vim.notify(output.stdout)
end, {nargs = '*'})
```

TL;DR – What you should do in November 2025

You are...	Do this
C die-hard, want true new vim.api	Write a libnvim shared library (example above)
Want max speed + modern tooling	Switch to Rust + nvim-oxi (5 minutes to get started)
Just need fast backend, not new APIs	Lua plugin + call your C binary with vim.system OR vim.uv.spawn
Want to suffer for art	Implement full MessagePack-RPC host in C (possible but nobody does it anymore)

Let me know which route excites you, and I'll give you a complete ready-to-compile repository template (C libnvim version or oxi version). The era of writing MessagePack by hand in C is pretty much over in 2025 unless you're doing it for fun or learning. 🚀

Below are the **realistic, battle-tested ways in 2025** to build Neovim plugins that are **driven by a C (or C++) backend** — excluding the two we already discussed (remote-plugin MessagePack-RPC host and libnvim shared-library extension).

#	Plugin Type	How the C backend is used	Performance	Complexity	When you'd actually choose this (2025 real-world examples)
1	<b>LuaJIT FFI plugin</b> (pure Lua + call into a .so written in C)	Your plugin is 99% Lua. Heavy lifting is done by a tiny hand-written .so (or .dll) that you call directly with ffi.cdef and ffi.load.	Near-native, zero serialization overhead	★★★★☆ (very easy)	Treesitter parsers before oxi existed, vim-clap, fzy-native, telescope-fzy-native, many fuzzy finders

2	<b>Lua module built with LuaRocks + C code</b> (require "myplugin")	You package a proper Lua C module (luaopen_myplugin) and install it system-wide or bundled with the plugin. Users just require("myplugin").	Native	★★★★☆	gitsigns.nvim's diff parts (early versions), luafilesystem, many legacy plugins
3	<b>Standalone CLI tool invoked with vim.system / vim.uv.spawn</b> (most popular in 2025)	Pure Lua plugin, your C program is a normal executable. Neovim spawns it synchronously or asynchronously.	Very fast (only pipe overhead)	★★★★☆ (easiest)	silicon.nvim, glow.nvim, image.nvim, copilot.vim alternatives, rg/fd/ugrep wrappers, llm plugins that call local ollama binary, etc.
4	<b>Tree-sitter parser written in C</b> (official way for new grammars)	You write a Tree-sitter grammar where parser.c is pure C. Neovim loads it automatically via nvim-treesitter.	Blazing fast parsing	★★★★☆	Every single new treesitter parser since 2021 (zig, rust, go, etc.)
5	<b>Libuv job + shared C library via dlopen + function pointers</b> (rare but powerful)	You dlopen your own .so from Lua using ffi.C.dlopen, cache function pointers, and call them from uv timers/jobs.	Native	★★★★☆	Used when you need to poll hardware or run a long-lived C daemon inside Neovim's process without blocking the event loop

### The Top 3 You'll Actually Want in 2025

#### 1. LuaJIT FFI + Tiny C Library (2025 sweet spot for most people who love C)

This is the **de-facto standard** when you want C speed but Lua ergonomics.

#### Directory layout

```
my-fast-finder/  
├─ lua/  
│   └─ myfinder.lua  
└─ src/  
    └─ matcher.c    → compiles to libmatcher.so
```

**matcher.c**

```
// src/matcher.c
#include <string.h>
#include <stdbool.h>

typedef struct { char* str; int len; } Slice;

bool fuzzy_match(const char* pattern, int plen, const char* haystack, int hlen, int*
score_out) {
    // your ultra-fast fuzzy algorithm in pure C
    // example: simple lowercase skip-match
    int score = 0;
    int pi = 0;
    for (int hi = 0; hi < hlen && pi < plen; ++hi) {
        if (tolower(haystack[hi]) == tolower(pattern[pi])) {
            score += 100 - pi*2;
            ++pi;
        }
    }
    if (pi == plen) {
        *score_out = score;
        return true;
    }
    return false;
}
```

**Compile (put in Makefile or just a one-liner)**

```
gcc -shared -fPIC -O3 -o lua/myfinder/libmatcher.so src/matcher.c
```

**lua/myfinder.lua**

```
local ffi = require("ffi")

ffi.cdef[[
bool fuzzy_match(const char* pattern, int plen, const char* haystack, int hlen, int*
score_out);
]]

local lib = ffi.load("myfinder/libmatcher.so")

local M = {}

function M.score_candidates(pattern, candidates)
    local pat = pattern:lower()
    local results = {}
    for _, cand in ipairs(candidates) do
        local hay = cand:lower()
        local score = ffi.new("int[1]")
        if lib.fuzzy_match(pat, #pat, hay, #hay, score) then
            table.insert(results, {candidate = cand, score = score[0]})
        end
    end
end
```

```

        end
    end
    table.sort(results, function(a,b) return a.score > b.score end)
    return results
end

return M

```

Usage in Telescope, fzf.vim, etc.: just `require("myfinder")` — feels 100% native and is 10-50× faster than pure Lua implementations.

### 2. Standalone C CLI + vim.system (the path of least resistance)

90% of new performance plugins in 2025 do exactly this.

Example: a grep replacement that's 3× faster than ripgrep in some cases.

```

// mygrep.c → compiled to ~/.local/bin/mygrep
int main(int argc, char** argv) {
    // parse args, search files blazingly fast, print JSON lines to stdout
}

```

Lua side:

```

local function mygrep(query)
    local job = vim.system(
        { "mygrep", query, vim.uv.cwd() },
        { text = true, stdout = vim.uv.new_pipe() }
    )
    job:wait()
    return vim.json.decode("[ " .. job:result().stdout:gsub("\n{", ",{") .. "]")
end

```

Zero plugin boilerplate, trivial debugging, works on Windows/macOS/Linux.

### 3. Tree-sitter parser in C (if you're adding syntax highlighting/querying)

Just follow the official template: <https://github.com/nvim-treesitter/tree-sitter->

Your `parser.c` is pure C, compiled with tree-sitter CLI → `parser` file loaded automatically by nvim-treesitter. This is literally how 400+ languages are supported.

### Verdict – Which one should you pick right now?

You want...	Choose...
Maximum speed + Lua-like API	LuaJIT FFI + tiny .so
Zero hassle, easy distribution	Standalone C CLI + vim.system
New language support / queries	Tree-sitter C parser
You still want to write mostly C	FFI route (you write the .so, Lua is glue)

Drop me a one-liner about what your plugin actually needs to do (fuzzy finding? LSP features? image processing? math? crypto?), and I'll give you a complete copy-paste repository skeleton for the best-fitting approach above. No more MessagePack suffering required in 2025. 🚀

Get ready — here are **real, mind-blowing projects and techniques** from 2025 that use **pure C (or C interop) as the true backend** for Neovim plugins. These are the ones that make people drop their jaw and go “wait, you can do THAT in Neovim with C?!”.

#	Project / Technique	What the C backend actually does	Why it's insanely cool in 2025	Link / Status
1	<b>image.nvim</b> (display images, PDFs, videos inside Neovim)	Kitty/Überzug/Sixel/UEBERZUG++ are all C daemons; the plugin just pipes raw pixel data from a tiny C converter (ffmpeg → raw RGBA) directly into the terminal protocol	You can literally view PDFs, GIFs, and webcam feeds inside Neovim buffers. Pure C speed for frame decoding.	<a href="https://github.com/3rd/image.nvim">https://github.com/3rd/image.nvim</a>
2	<b>silicon.nvim</b> (beautiful code screenshots)	Spawns a headless Neovim instance + a custom C program that uses <b>resvg</b> (Rust → C interop) + <b>fontdue</b> + raw Cairo to render code at 4K with perfect ligatures in <50ms	Faster and sharper than VSCode screenshots. The C/Rust binary does 99% of the work.	<a href="https://github.com/kriva/silicon">https://github.com/kriva/silicon</a> (on nvim wrapper)
3	<b>nvim-gdb &amp; CodeLLDB</b>	The actual debugger engine (lldb-vscode) is a 100% C++ monolith. The Neovim plugin just sends JSON-RPC to a tiny C++ DAP server compiled as a standalone binary	Full-featured debugging with breakpoints, watch, threads — feels like VSCode but in Neovim	<a href="https://github.com/mfussenegger">https://github.com/mfussenegger</a> CodeLLDB binary



4	<b>helix-gpui</b> experiments (2024-2025)	People are embedding <b>gpui</b> (Zed editor's GPU renderer written in C++) as a shared library and rendering rich text directly into Neovim windows via external windows + overlay hacks	Actual GPU-accelerated text rendering inside Neovim (60–240 FPS even with 100k lines)	Proof-of-concept: search “neovim” on GitHub
5	<b>neovim + libffmpeg</b> (video playback inside Neovim)	A C program using libavcodec/libavformat decodes H.264/MP4 frames → sends raw RGBA frames over a Unix socket → Neovim plugin renders them with kitty graphics protocol at 60 FPS	You can watch YouTube or local videos inside Neovim. Yes, really.	<a href="https://github.com/niwavi/niwavi">https://github.com/niwavi/niwavi</a> , <a href="https://github.com/edluffy/hologr">https://github.com/edluffy/hologr</a> (spiritual successor)
6	<b>nvim-snappy</b> (real-time collaborative editing like Google Docs)	Backend is a tiny C daemon using <b>libsodium</b> + <b>cJSON</b> + <b>y-sync</b> CRDT algorithm for sub-millisecond conflict-free merges	Multiple people typing in the same buffer with zero lag and perfect convergence	Early 2025 prototype: <a href="https://github.com/y-sync/y-sync">https://github.com/y-sync/y-sync</a> + Neovim frontend
7	<b>neovim + llama.cpp</b> (local 70B LLM inside Neovim)	llama.cpp is pure C/C++. Plugins like <b>gp.nvim</b> , <b>avante.nvim</b> , <b>codecompanion.nvim</b> just spawn the binary and stream tokens over stdin/stdout	You get 30–50 tokens/sec on a laptop, fully offline, with perfect Neovim integration (inline completions, chat panels, etc.)	<a href="https://github.com/YetAnotherGrc">https://github.com/YetAnotherGrc</a> (2025's hottest)
8	<b>neovim + libtorch</b> (run PyTorch models directly from C)	Someone compiled a minimal libtorch .so and loads it via FFI → runs Stable Diffusion 1.5 in ~1.2s per image inside Neovim	Generate images from prompts inside a Neovim buffer. Yes, you read that right.	<a href="https://github.com/yacineMTB/diffusion">https://github.com/yacineMTB/diffusion</a> experimental nvim-sd plugin
9	<b>neovim + SQLite + libsql</b> (full	The entire SQLite engine is C. Plugins like <b>vim-dadbod</b> + <b>sqlite.lua</b> + custom C	Turn Neovim into a full-fledged data	<a href="https://github.com/tami5/sqlite.lua">https://github.com/tami5/sqlite.lua</a>

	SQL database inside your editor)	extensions let you run analytical queries on millions of rows inside Neovim	notebook (better than Obsidian Logseq for big CSVs)	
10	<b>neovide</b> (the GPU renderer itself)	Neovide is literally a C/Rust frontend that embeds the entire Neovim C core as a static library and renders with wgpu/OpenGL/Metal	Smooth scroll, font ligatures at 240 FPS, animated floats, cursor effects — all powered by the same Neovim C core you already have	<a href="https://neovide.dev">https://neovide.dev</a> — basically "V Neovim under the hood"

## The Ultimate 2025 Inspiration Stack (copy-paste ready)

Want to blow minds? Combine these into one monster plugin:

```
// my_ultimate_backend.c → compiled as ultimate.so + ultimate_cli
- llama.cpp → local LLM
- ffmpeg → video decoding
- sqlite → embedded DB
- libsodium → encryption
- resvg → SVG rendering
- tree-sitter parsers → 100+ languages
```

Lua/FFI frontend spawns or FFI-calls them → you get:

- AI code completion
- Inline image/PDF/video preview
- Encrypted notes with live collab
- 4K code screenshots
- Full-text search over 10M lines in <10ms

All inside Neovim. All powered by battle-tested C libraries.

That's the real power move in 2025: **Neovim is no longer "just a text editor" — it's the world's most hackable application platform, and C is the rocket fuel.**

Now go build something that makes people say "wait... that's Neovim?!" 🔥

What's your next insane idea? I'll help you scaffold the C part in 5 minutes.

---

# with-mistral

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 a C backend is a powerful way to extend Neovim's functionality, especially for performance-critical tasks. Here's a step-by-step guide to help you get started:

---

## 1. Understand the Architecture

Neovim supports plugins written in Lua, Vimscript, and remote plugins (using RPC). For a C backend, you'll use Neovim's **remote plugin** feature, which allows you to communicate with Neovim via its **msgpack-RPC API**.

- **Frontend:** Lua/Vimscript (handles user interaction, UI, and Neovim API calls).
  - **Backend:** C program (handles heavy lifting, logic, or performance-critical tasks).
- 

## 2. Set Up the Project Structure

Organize your project like this:

```
/myplugin/  
├─ lua/  
│   └─ myplugin.lua      # Lua frontend  
├─ src/  
│   └─ myplugin.c        # C backend  
├─ Makefile              # Build script  
└─ plugin/myplugin.vim    # Vimscript entry point (optional)
```

---

## 3. Write the C Backend

Your C program will communicate with Neovim via **stdin/stdout** using **msgpack-RPC**. Neovim expects messages in a specific format.

### Example: Simple C Backend

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <msgpack.h>  
  
void handle_request(msgpack_object *request) {  
    // Parse the request and send a response  
    msgpack_sbuffer sbuf;  
    msgpack_packer pk;  
    msgpack_sbuffer_init(&sbuf);
```

```

msgpack_packer_init(&pk, &sbuf, msgpack_sbuffer_write);

// Example: Echo the request
msgpack_pack_array(&pk, 4);
msgpack_pack_uint64(&pk, 0); // msgid (0 for notification)
msgpack_pack_str(&pk, 11);
msgpack_pack_str_body(&pk, "notification", 11);
msgpack_pack_str(&pk, 4);
msgpack_pack_str_body(&pk, "echo", 4);
msgpack_pack_str(&pk, 5);
msgpack_pack_str_body(&pk, "hello", 5);

fwrite(sbuf.data, 1, sbuf.size, stdout);
msgpack_sbuffer_destroy(&sbuf);
}

int main() {
    msgpack_unpacked result;
    msgpack_unpack_return ret;
    msgpack_unpacked_init(&result);

    while ((ret = msgpack_unpack_next(&result, stdin, NULL)) ==
MSGPACK_UNPACK_SUCCESS) {
        handle_request(&result.data);
    }

    msgpack_unpacked_destroy(&result);
    return 0;
}

```

## Key Points

- Use the [msgpack-c](#) library to serialize/deserialize messages.
- Neovim sends requests as **msgpack arrays** and expects responses in the same format.
- The first element of the array is the **message type** (0 for notification, 1 for request, 2 for response).

## 4. Write the Lua Frontend

The Lua frontend will:

- Register your plugin with Neovim.
- Start the C process and handle communication.

### Example: Lua Frontend

```

local function start_backend()
    local handle = io.popen("./myplugin", "r+")
    local function send_request(method, args)
        local request = {
            { 0, "notification", method, args }
        }
    end
end

```

```

    local msg = vim.fn.json_encode(request)
    handle:write(msg .. "\n")
    handle:flush()
end

local function read_response()
    local response = handle:read("*l")
    return vim.fn.json_decode(response)
end

return {
    send_request = send_request,
    read_response = read_response,
}
end

local backend = start_backend()

-- Example: Call the C backend
backend.send_request("echo", { "hello" })
local response = backend.read_response()
print(vim.inspect(response))

```

### Key Points

- Use `io.popen` to start the C process.
- Communicate via **stdin/stdout**.
- Use `vim.fn.json_encode` / `vim.fn.json_decode` for simplicity (or use msgpack directly).

## 5. Register the Plugin

Add a Vimscript file to load your Lua plugin:

```

" plugin/myplugin.vim
lua require('myplugin')

```

## 6. Build and Test

### Build the C Program

```
gcc -o myplugin src/myplugin.c -lmsgpackc
```

### Test in Neovim

1. Place the plugin in your Neovim runtimepath (e.g., `~/.config/nvim/`).
2. Start Neovim and test:

```
:lua require('myplugin').test()
```

---

## 7. Debugging Tips

- Use `socat` to manually test the C program:

```
socat - UNIX-LISTEN:/tmp/nvim.fifo
```

- Log messages in C using `fprintf(stderr, ...)`.
- Use `:messages` in Neovim to check for errors.

---

## 8. Resources

- [Neovim Remote Plugin Guide](#)
- [msgpack-c](#)
- [Neovim Lua Guide](#)

---

Would you like a deeper dive into any specific part, such as handling async communication or optimizing performance?

---

If your goal is to **enrich Neovim's APIs** (i.e., add new Lua functions or extend existing ones) using a C backend, you'll want to create a **Neovim remote plugin** that exposes new Lua-callable functions. This approach allows you to write high-performance or system-level code in C, while making it accessible from Lua in Neovim.

Here's a step-by-step guide tailored for this use case:

---

## 1. Overview of the Approach

- **C Backend:** Implements the core logic or system-level functionality.
- **Lua Frontend:** Registers new Lua functions that call the C backend via RPC.
- **Neovim RPC:** Uses Neovim's msgpack-RPC API for communication.

---

## 2. Project Structure

```
/myplugin/  
├─ lua/  
│   └─ myplugin.lua      # Lua frontend (registers new APIs)  
├─ src/  
│   └─ myplugin.c        # C backend (implements the logic)  
├─ Makefile              # Build script  
└─ plugin/myplugin.vim    # Vimscript entry point (optional)
```

---

## 3. Write the C Backend

Your C program should:

- Listen for RPC requests from Neovim.
- Implement the new functionality (e.g., system calls, heavy computation).
- Return results in msgpack format.

### Example: C Backend (Exposing a New API)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <msgpack.h>

void handle_request(msgpack_object *request) {
    msgpack_object_array *arr = &request->via.array;
    if (arr->size >= 3) {
        msgpack_object *method = &arr->ptr[2];
        if (method->type == MSGPACK_OBJECT_STR) {
            const char *method_str = method->via.str.ptr;
            if (strcmp(method_str, "my_new_api") == 0) {
                // Implement your new API logic here
                msgpack_sbuffer sbuf;
                msgpack_packer pk;
                msgpack_sbuffer_init(&sbuf);
                msgpack_packer_init(&pk, &sbuf, msgpack_sbuffer_write);

                // Example: Return a string
                msgpack_pack_array(&pk, 4);
                msgpack_pack_uint64(&pk, 1); // Response msgid
                msgpack_pack_str(&pk, 7);
                msgpack_pack_str_body(&pk, "response", 7);
                msgpack_pack_str(&pk, 9);
                msgpack_pack_str_body(&pk, "my_new_api", 9);
                msgpack_pack_str(&pk, 12);
                msgpack_pack_str_body(&pk, "Hello from C!", 12);

                fwrite(sbuf.data, 1, sbuf.size, stdout);
                msgpack_sbuffer_destroy(&sbuf);
            }
        }
    }
}

int main() {
    msgpack_unpacked result;
    msgpack_unpacked_init(&result);
    msgpack_unpack_return ret;

    while ((ret = msgpack_unpack_next(&result, stdin, NULL)) ==
MSGPACK_UNPACK_SUCCESS) {
        handle_request(&result.data);
    }
}
```

```
msgpack_unpacked_destroy(&result);  
return 0;  
}
```

### Key Points

- The C program reads msgpack requests from stdin.
  - It checks the method name and implements the logic for each API.
  - It writes msgpack responses to stdout.
- 

## 4. Write the Lua Frontend

The Lua frontend registers new Lua functions that call the C backend.

### Example: Lua Frontend (Registering a New API)

```
local function start_backend()  
    local handle = io.popen("./myplugin", "r+")  
    local function send_request(method, args, callback)  
        local request = {  
            { 1, "request", method, args }  
        }  
        local msg = vim.fn.json_encode(request)  
        handle:write(msg .. "\n")  
        handle:flush()  
  
        local response = handle:read("*l")  
        local ok, result = pcall(vim.fn.json_decode, response)  
        if ok and callback then  
            callback(result)  
        end  
    end  
end  
  
return {  
    send_request = send_request,  
}  
end  
  
local backend = start_backend()  
  
-- Expose the new API to Lua  
local M = {}  
  
function M.my_new_api(callback)  
    backend.send_request("my_new_api", {}, callback)  
end  
  
return M
```

### Key Points



- `send_request` sends a msgpack request to the C backend.
  - The Lua function `my_new_api` is callable from Neovim Lua scripts.
  - The callback receives the response from the C backend.
- 

## 5. Register the Plugin

Add a Vimscript file to load your Lua plugin:

```
" plugin/myplugin.vim
lua require('myplugin')
```

---

## 6. Build and Test

### Build the C Program

```
gcc -o myplugin src/myplugin.c -lmsgpackc
```

### Test in Neovim

1. Place the plugin in your Neovim runtimepath (e.g., `~/.config/nvim/`).
2. Start Neovim and test:

```
:lua require('myplugin').my_new_api(function(result)
print(vim.inspect(result)) end)
```

---

## 7. Debugging Tips

- Use `socat` to manually test the C program:

```
socat - UNIX-LISTEN:/tmp/nvim.fifo
```

- Log messages in C using `fprintf(stderr, ...)`.
  - Use `:messages` in Neovim to check for errors.
- 

## 8. Advanced: Async and Performance

- Use Neovim's `vim.schedule` for async callbacks.
  - For performance-critical APIs, minimize serialization overhead.
- 

## 9. Resources

- [Neovim Remote Plugin Guide](#)
  - [msgpack-c](#)
  - [Neovim Lua Guide](#)
-

## Next Steps

- Extend the C backend to implement more APIs.
- Add error handling and input validation.
- Optimize for performance if needed.

Would you like to focus on a specific API or use case (e.g., filesystem operations, system calls, or data processing)?

---

If you want to explore **another type of Neovim plugin** that still uses a **C program as the backend**, but with a different architecture, consider building a **Neovim UI plugin** or a **Neovim TUI (Text User Interface) extension**. These plugins can enhance Neovim's interface, add custom UI elements, or even create entirely new visual experiences—all while leveraging the performance and system-level access of C.

---

## Option: Neovim TUI Plugin with C Backend

This approach involves creating a **custom TUI (Text User Interface) plugin** that interacts with Neovim's core via its API, but renders UI elements (e.g., popups, status bars, or sidebars) using a C program. This is useful for:

- Custom statuslines or tab bars.
- Interactive dashboards or sidebars.
- Terminal-based UI extensions.

---

### 1. Architecture Overview

- **C Backend**: Handles rendering, user input, and system-level tasks.
  - **Lua Frontend**: Communicates with Neovim's API and the C backend via RPC.
  - **Neovim**: Acts as the host, providing the editing environment and API access.
- 

### 2. Project Structure

```
/my_tui_plugin/
├─ lua/
│   └─ my_tui_plugin.lua # Lua frontend (handles Neovim API and RPC)
├─ src/
│   └─ my_tui_plugin.c   # C backend (handles rendering and input)
├─ Makefile              # Build script
└─ plugin/my_tui_plugin.vim # Vimscript entry point (optional)
```

---

### 3. Write the C Backend

The C program will:

- Render UI elements (e.g., using `ncurses` or a custom TUI library).
- Listen for user input and send updates to Neovim.
- Communicate with Neovim via **stdin/stdout** using msgpack-RPC.

#### Example: C Backend (TUI Rendering)

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <msgpack.h>
#include <ncurses.h> // For TUI rendering

void init_tui() {
    initscr();
    cbreak();
    noecho();
    keypad(stdscr, TRUE);
}

void render_ui(const char *text) {
    clear();
    mvprintw(0, 0, "%s", text);
    refresh();
}

void handle_request(msgpack_object *request) {
    msgpack_object_array *arr = &request->via.array;
    if (arr->size >= 3) {
        msgpack_object *method = &arr->ptr[2];
        if (method->type == MSGPACK_OBJECT_STR) {
            const char *method_str = method->via.str.ptr;
            if (strcmp(method_str, "render") == 0) {
                // Extract the text to render from the request
                msgpack_object *args = &arr->ptr[3];
                if (args->type == MSGPACK_OBJECT_ARRAY) {
                    msgpack_object_array *args_arr = &args->via.array;
                    if (args_arr->size >= 1) {
                        msgpack_object *text_obj = &args_arr->ptr[0];
                        if (text_obj->type == MSGPACK_OBJECT_STR) {
                            render_ui(text_obj->via.str.ptr);
                        }
                    }
                }
            }
        }
    }
}

int main() {
    init_tui();
    msgpack_unpacked result;
    msgpack_unpacked_init(&result);
    msgpack_unpack_return ret;

    while ((ret = msgpack_unpack_next(&result, stdin, NULL)) ==
MSGPACK_UNPACK_SUCCESS) {
        handle_request(&result.data);
    }
}

```

```

    }

    endwin();
    msgpack_unpacked_destroy(&result);
    return 0;
}

```

### Key Points

- Uses `ncurses` for rendering a TUI.
- Listens for RPC requests from Neovim and updates the UI accordingly.
- Sends user input back to Neovim via msgpack-RPC.

## 4. Write the Lua Frontend

The Lua frontend will:

- Register the plugin with Neovim.
- Start the C process and handle communication.
- Expose Lua functions to update the TUI.

### Example: Lua Frontend (TUI Plugin)

```

local function start_tui_backend()
    local handle = io.popen("./my_tui_plugin", "r+")
    local function send_request(method, args, callback)
        local request = {
            { 1, "request", method, args }
        }
        local msg = vim.fn.json_encode(request)
        handle:write(msg .. "\n")
        handle:flush()

        if callback then
            local response = handle:read("*l")
            local ok, result = pcall(vim.fn.json_decode, response)
            if ok then
                callback(result)
            end
        end
    end
end

return {
    send_request = send_request,
}

end

local tui = start_tui_backend()

-- Expose a Lua function to update the TUI
local M = {}

```

```
function M.render(text)
    tui.send_request("render", { text }, function(result)
        -- Handle response if needed
    end)
end

return M
```

### Key Points

- Starts the C backend process.
  - Exposes a `render` function to update the TUI from Lua.
  - Uses callbacks for async communication.
- 

## 5. Register the Plugin

Add a Vimscript file to load your Lua plugin:

```
" plugin/my_tui_plugin.vim
lua require('my_tui_plugin')
```

## 6. Build and Test

### Build the C Program

```
gcc -o my_tui_plugin src/my_tui_plugin.c -lmsgpackc -lcurses
```

### Test in Neovim

1. Place the plugin in your Neovim runtimepath (e.g., `~/.config/nvim/`).
2. Start Neovim and test:

```
:lua require('my_tui_plugin').render("Hello from Neovim!")
```

## 7. Debugging Tips

- Use `strace` or `gdb` to debug the C program.
  - Log messages in C using `fprintf(stderr, ...)`.
  - Use `:messages` in Neovim to check for errors.
- 

## 8. Advanced Features

- **Async Updates:** Use `vim.schedule` to avoid blocking Neovim.
  - **Custom Events:** Emit Neovim events from the C backend.
  - **Performance:** Optimize rendering and minimize IPC overhead.
- 

## 9. Resources

- [Neovim Remote Plugin Guide](#)

- [ncurses Library](#)
- [msgpack-c](#)

---

## Why This Approach?

- **Performance:** C is ideal for rendering and system-level tasks.
- **Flexibility:** You can create custom UI elements not possible with pure Lua/Vimscript.
- **Integration:** Seamlessly interacts with Neovim's API and Lua ecosystem.

---

## Next Steps

- Extend the TUI with more interactive elements (e.g., menus, progress bars).
- Add support for mouse input or custom keybindings.
- Explore integrating with Neovim's built-in LSP or DAP for advanced features.

---

Here are **three cutting-edge, inspirational ideas** for using a **C program as a backend for Neovim Lua plugins**. These ideas push the boundaries of what's possible with Neovim, leveraging the performance, system access, and low-level control of C to create powerful, unique, and innovative plugins.

---

## 1. Real-Time System Monitor Dashboard

**Concept:** A Neovim plugin that displays a **real-time system monitor dashboard** (CPU, memory, network, disk I/O, processes) as a floating window or sidebar. The C backend collects system metrics, while the Lua frontend renders them in Neovim.

---

### Why C?

- Direct access to system APIs ( `/proc` , `sysctl` , `libproc` , etc.).
- High-performance polling and data aggregation.
- Minimal overhead for real-time updates.

---

### C Backend Features

- Poll system metrics (CPU usage, memory, network stats, etc.).
- Aggregate and format data for Neovim.
- Use lightweight libraries like `libuv` for async I/O.

### Example C Snippet (System Metrics)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/sysinfo.h>
#include <msgpack.h>

void get_cpu_usage(double *usage) {
    // Read CPU stats from /proc/stat
    FILE *fp = fopen("/proc/stat", "r");
    // Parse and calculate CPU usage
    *usage = 0.42; // Example value
}
```

```

    fclose(fp);
}

void send_metrics() {
    struct sysinfo info;
    sysinfo(&info);

    double cpu_usage;
    get_cpu_usage(&cpu_usage);

    msgpack_sbuffer sbuf;
    msgpack_packer pk;
    msgpack_sbuffer_init(&sbuf);
    msgpack_packer_init(&pk, &sbuf, msgpack_sbuffer_write);

    msgpack_pack_map(&pk, 3);
    msgpack_pack_str(&pk, 3);
    msgpack_pack_str_body(&pk, "cpu", 3);
    msgpack_pack_double(&pk, cpu_usage);

    msgpack_pack_str(&pk, 4);
    msgpack_pack_str_body(&pk, "mem", 3);
    msgpack_pack_uint64(&pk, info.totalram - info.freeram);

    msgpack_pack_str(&pk, 4);
    msgpack_pack_str_body(&pk, "uptime", 6);
    msgpack_pack_uint64(&pk, info.uptime);

    fwrite(sbuf.data, 1, sbuf.size, stdout);
    msgpack_sbuffer_destroy(&sbuf);
}

int main() {
    while (1) {
        send_metrics();
        sleep(1); // Update every second
    }
    return 0;
}

```

## Lua Frontend Features

- Render metrics in a floating window or sidebar.
- Update the display in real-time using Neovim's `vim.schedule`.
- Allow user interaction (e.g., sorting processes, killing tasks).

### Example Lua Snippet (Rendering)

```

local function start_monitor()
    local handle = io.popen("./system_monitor", "r")
    local function update_display()

```

```
local data = handle:read("*l")
local metrics = vim.fn.json_decode(data)
-- Render metrics in a floating window
vim.api.nvim_echo({{string.format("CPU: %.2f%%, Mem: %dMB", metrics.cpu *
100, metrics.mem / 1024), "Normal"}}}, true, {})
end
vim.loop.new_timer():start(0, 1000, vim.schedule_wrap(update_display))
end

start_monitor()
```

---

## Inspiration

- Integrate with Neovim's built-in terminal for interactive process management.
  - Add alerts for high resource usage.
  - Support customizable layouts and themes.
- 

## 2. Neovim-Powered Game Engine

**Concept:** Turn Neovim into a **retro game engine** or interactive fiction platform. The C backend handles game logic, physics, and rendering, while Lua scripts define game behavior and assets.

---

### Why C?

- High-performance game loop and physics.
  - Direct access to terminal rendering (e.g., `ncurses`, ANSI escape codes).
  - Portability and low-level control.
- 

### C Backend Features

- Game loop and state management.
- Collision detection and physics.
- Rendering using ANSI escape codes or `ncurses`.

### Example C Snippet (Game Loop)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <msgpack.h>

typedef struct {
    int x, y;
} Player;

void game_loop() {
    Player player = {10, 10};
    while (1) {
        // Update game state
        player.x += 1;
```



```

    // Render game state
    printf("\033[2J\033[H"); // Clear screen
    printf("Player: (%d, %d)\n", player.x, player.y);

    // Send state to Neovim
    msgpack_sbuffer sbuf;
    msgpack_packer pk;
    msgpack_sbuffer_init(&sbuf);
    msgpack_packer_init(&pk, &sbuf, msgpack_sbuffer_write);

    msgpack_pack_map(&pk, 2);
    msgpack_pack_str(&pk, 1);
    msgpack_pack_str_body(&pk, "x", 1);
    msgpack_pack_int(&pk, player.x);

    msgpack_pack_str(&pk, 1);
    msgpack_pack_str_body(&pk, "y", 1);
    msgpack_pack_int(&pk, player.y);

    fwrite(sbuf.data, 1, sbuf.size, stdout);
    fflush(stdout);

    usleep(100000); // 100ms delay
}
}

int main() {
    game_loop();
    return 0;
}

```

## Lua Frontend Features

- Define game assets, levels, and scripts.
- Handle user input and send commands to the C backend.
- Render game state in Neovim buffers or floating windows.

### Example Lua Snippet (Game Integration)

```

local function start_game()
    local handle = io.popen("./neovim_game", "r+")
    local function update_game()
        local data = handle:read("*l")
        local state = vim.fn.json_decode(data)
        -- Render game state in a buffer
        vim.api.nvim_buf_set_lines(0, 0, -1, false, {string.format("Player: (%d, %d)", state.x, state.y)})
    end
    vim.loop.new_timer():start(0, 100, vim.schedule_wrap(update_game))
end

```

```
start_game()
```

---

## Inspiration

- Create a **Neovim-based roguelike** or puzzle game.
  - Add multiplayer support using Neovim's RPC.
  - Integrate with Neovim's LSP for scripting game logic in Lua.
- 

## 3. Neovim as a Music Tracker (Chiptune Editor)

**Concept:** Build a **music tracker** (like FastTracker or Renoise) inside Neovim. The C backend handles audio synthesis and playback, while the Lua frontend provides a tracker-style UI for editing notes and patterns.

---

### Why C?

- Real-time audio synthesis and playback.
  - Low-latency MIDI or audio output.
  - Efficient sample and pattern management.
- 

### C Backend Features

- Audio synthesis using `libao`, `portaudio`, or `SDL`.
- MIDI input/output support.
- Sample and pattern management.

### Example C Snippet (Audio Playback)

```
#include <stdio.h>
#include <stdlib.h>
#include <portaudio.h>
#include <msgpack.h>

#define SAMPLE_RATE 44100

void audio_callback(const void *input, void *output, unsigned long frames,
                   const PaStreamCallbackTimeInfo *timeInfo, PaStreamCallbackFlags
statusFlags,
                   void *userData) {
    float *out = (float *)output;
    for (unsigned long i = 0; i < frames; i++) {
        *out++ = 0.5f * sin(2 * M_PI * 440.0f * i / SAMPLE_RATE); // 440Hz sine wave
    }
}

void start_audio() {
    Pa_Initialize();
    PaStream *stream;
    Pa_OpenDefaultStream(&stream, 0, 1, paFloat32, SAMPLE_RATE, 256, audio_callback,
NULL);
```

```

Pa_StartStream(stream);

// Send audio status to Neovim
msgpack_sbuffer sbuf;
msgpack_packer pk;
msgpack_sbuffer_init(&sbuf);
msgpack_packer_init(&pk, &sbuf, msgpack_sbuffer_write);

msgpack_pack_map(&pk, 1);
msgpack_pack_str(&pk, 4);
msgpack_pack_str_body(&pk, "play", 4);
msgpack_pack_bool(&pk, 1);

fwrite(sbuf.data, 1, sbuf.size, stdout);
msgpack_sbuffer_destroy(&sbuf);

while (1) sleep(1); // Keep running
}

int main() {
    start_audio();
    return 0;
}

```

---

## Lua Frontend Features

- Render a tracker-style UI in Neovim buffers.
- Edit notes, patterns, and samples.
- Send playback commands to the C backend.

### Example Lua Snippet (Tracker UI)

```

local function start_tracker()
    local handle = io.popen("./neovim_tracker", "r+")
    local function play_note(note)
        handle:write(string.format('{ "command": "play", "note": "%s"}\n', note))
    end
    -- Render tracker UI and bind keys to play_note
    vim.api.nvim_set_keymap('n', '<leader>p', ':lua play_note("C4")<CR>', {noremap =
true})
end

start_tracker()

```

---

## Inspiration

- Add **VST plugin support** for advanced synthesis.
  - Integrate with **Neovim's terminal** for live coding music.
  - Create a **collaborative music editing** mode using Neovim's RPC.
-

## Final Thoughts

These ideas showcase how you can **push Neovim beyond traditional editing** by combining its Lua ecosystem with the power of C. Whether you're building a **system monitor**, **game engine**, or **music tracker**, the combination of C and Lua opens up endless possibilities for innovation.

**Which idea excites you the most?** Would you like to dive deeper into one of these concepts or explore another creative use case?

---

# with-ms-copilot

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

---

Perfect challenge, Fudzer ⚡. You want a **Neovim plugin** where the backend logic is written in **C**, and Lua acts as the ergonomic glue. Let's break this down into a blueprint you can expand modularly:

---

## Architecture Overview



- **C backend:** Implements the heavy logic (e.g., parsing, computation, system calls).
  - **Lua frontend:** Exposes ergonomic commands, toggles, and wrappers inside Neovim.
  - **Bridge:** Use Neovim's **remote plugin API** or **Lua FFI** to connect Lua ↔ C.
- 

## Integration Options

### 1. LuaJIT FFI (Fast & Direct)

- Load your compiled C library ( `.so` / `.dll` ) directly in Lua.
- Example:



```
local ffi = require("ffi")
ffi.cdef[[
    int add(int a, int b);
]]
local lib = ffi.load("myplugin")
print(lib.add(2, 3)) -- 5
```

-  Pros: Fast, minimal overhead, easy to prototype.
-  Cons: Must manage ABI compatibility carefully.

### 2. Neovim Remote Plugin (via msgpack-rpc)

- Write a C program that speaks Neovim's RPC protocol.
- Register functions in `rpplugin/` manifest.
- Example manifest ( `rpplugin.vim` ):

```
" Define remote plugin
call remote#host#Register('cplugin', '0', [])
```

-  Pros: Full Neovim integration, async jobs, works cross-platform.
  -  Cons: More boilerplate.
- 

## Project Skeleton

```
myplugin/
├─ lua/
│   └─ myplugin.lua    # Lua wrapper
├─ src/
│   └─ myplugin.c      # C backend
├─ Makefile            # Build C backend
└─ rplugin.vim         # Remote plugin manifest (if RPC)
```

---

## ⚙️ Example: Minimal C Backend + Lua Wrapper

### src/myplugin.c

```
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}
```

Compile:

```
gcc -shared -fPIC -o myplugin.so src/myplugin.c
```

### lua/myplugin.lua

```
local ffi = require("ffi")
ffi.cdef[[
    int add(int a, int b);
]]
local lib = ffi.load("myplugin")

local M = {}

function M.add_numbers(a, b)
    return lib.add(a, b)
end

vim.api.nvim_create_user_command("AddNumbers", function(opts)
    local args = vim.split(opts.args, " ")
    local result = M.add_numbers(tonumber(args[1]), tonumber(args[2]))
    print("Result: " .. result)
end, { nargs = 2 })

return M
```

Usage inside Neovim:

```
:AddNumbers 2 3
```

→ prints `Result: 5`

---



## Expansion Ideas

- **Parameterization:** Add toggles in Lua to switch between different C functions.
  - **Async jobs:** Use Neovim's `jobstart()` to run long-running C processes.
  - **Error handling:** Wrap C calls with Lua error checks for robustness.
  - **Cross-platform:** Provide `Makefile` targets for Linux/macOS/Windows.
- 



## Next Steps

1. Decide: **FFI vs RPC** (FFI is simpler, RPC is more “official”).
  2. Scaffold your C backend with modular functions.
  3. Wrap them in Lua commands, mappings, or autocommands.
  4. Add ergonomic toggles (your specialty 😊).
- 

Would you like me to **blueprint a full remote plugin (RPC)** example next, or keep iterating on the **FFI-based approach** for speed and minimalism?

---

Great pivot, Fudzer ⚡. If your goal is to **enrich Neovim's APIs themselves** (rather than just wrapping C logic), you're essentially building a **native extension** that plugs deeper into Neovim's core. Let's blueprint that:

---



## Approaches to Extending Neovim APIs

### 1. Remote Plugin via Msgpack-RPC

- Neovim exposes its API over **msgpack-rpc**.
- Your C backend can register new functions that behave like native Neovim API calls.
- Example flow:
  - Implement RPC handlers in C.
  - Register them in `rplugin.vim`.
  - Call them from Lua/Vimscript as if they were built-in.

This makes your plugin feel like it's **adding new API endpoints**.

---

### 2. Direct Embedding (libnvim)

- Neovim itself is written in C, and its core APIs are exposed in `src/nvim/api/`.
  - You can build against **libnvim** and expose new functionality.
  - This is closer to “patching” Neovim with custom C modules.
  - Heavyweight, but gives you **full control**.
- 

### 3. LuaJIT FFI + API Wrappers

- Use LuaJIT's FFI to call your C functions, but design them to **mirror Neovim's API style**.
  - Example: create `nvim_buf_transform()` in C, then expose it in Lua as if it were a native buffer API.
  - This is ergonomic and minimal, while still enriching the API surface.
- 



## Example Skeleton: Adding a Custom Buffer API

### src/myplugin.c

```
#include <string.h>
#include <stdlib.h>

// Example: reverse a string (pretend it's buffer content)
char* reverse(const char* input) {
    size_t len = strlen(input);
    char* out = malloc(len + 1);
    for (size_t i = 0; i < len; i++) {
        out[i] = input[len - i - 1];
    }
    out[len] = '\0';
    return out;
}
```

Compile:

```
gcc -shared -fPIC -o myplugin.so src/myplugin.c
```

### lua/myplugin.lua

```
local ffi = require("ffi")
ffi.cdef[[
    char* reverse(const char* input);
]]

local lib = ffi.load("myplugin")

local M = {}

function M.nvim_buf_reverse(bufnr)
    local lines = vim.api.nvim_buf_get_lines(bufnr, 0, -1, false)
    for i, line in ipairs(lines) do
        local rev = ffi.string(lib.reverse(line))
        lines[i] = rev
    end
    vim.api.nvim_buf_set_lines(bufnr, 0, -1, false, lines)
end

vim.api.nvim_create_user_command("BufReverse", function(opts)
    M.nvim_buf_reverse(0)
end, {})

return M
```

Usage:



```
:BufReverse
```

→ reverses all buffer lines, as if you had a new API call `nvim_buf_reverse`.

## Expansion Ideas

- **Custom buffer ops:** `nvim_buf_sort()`, `nvim_buf_filter()`.
- **Window APIs:** new layout managers in C, exposed as `nvim_win_grid()`.
- **Async jobs:** enrich `vim.loop` with C-powered workers.
- **Composable toggles:** expose ergonomic Lua wrappers for your C APIs (your specialty).

## Strategic Choice

- If you want **deep integration** (like contributing new official APIs), go with **libnvim embedding**.
- If you want **plugin-level enrichment**, FFI or RPC is cleaner and keeps things modular.

👉 Do you want me to **blueprint a full RPC-based plugin** that registers new API-like functions (so they appear in `:help api` style), or keep iterating on the **FFI approach** where you mimic Neovim's API ergonomics?

Alright Fudzer ⚡, let's explore another **plugin archetype** that still uses a **C backend**, but instead of just API enrichment or FFI wrappers, we'll pivot into a **service-style plugin**. This type of plugin treats your C program as a **background worker** that Neovim communicates with asynchronously.

## Type: Async Job Plugin (C Worker Process)

### Concept

- Your C program runs as a **standalone executable**.
- Neovim launches it via `jobstart()` or RPC.
- Communication happens through **stdin/stdout** using JSON or msgpack.
- This lets you offload heavy tasks (parsing, linting, compilation, AI inference, etc.) to C while keeping Neovim responsive.

## Project Skeleton

```
myplugin/
├─ lua/
│   └─ myplugin.lua    # Lua glue code
├─ src/
│   └─ worker.c        # C backend worker
├─ Makefile            # Build worker
└─ rplugin.vim         # Optional manifest
```

## Example: C Worker That Uppercases Text

```
src/worker.c
```

```

#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main() {
    char buffer[1024];
    while (fgets(buffer, sizeof(buffer), stdin)) {
        // Trim newline
        buffer[strcspn(buffer, "\n")] = 0;

        // Uppercase transform
        for (int i = 0; buffer[i]; i++) {
            buffer[i] = toupper(buffer[i]);
        }

        // Send back to Neovim
        printf("%s\n", buffer);
        fflush(stdout);
    }
    return 0;
}

```

Compile:

```
gcc -o worker src/worker.c
```

## lua/myplugin.lua

```

local M = {}

function M.start_worker()
    local job_id = vim.fn.jobstart({"/.worker"}, {
        on_stdout = function(_, data, _)
            if data then
                for _, line in ipairs(data) do
                    if line ~= "" then
                        print("Worker says: " .. line)
                    end
                end
            end
        end,
        on_exit = function(_, code, _)
            print("Worker exited with code " .. code)
        end,
    })
    M.job_id = job_id
end

function M.send_to_worker(text)

```

```

    if M.job_id then
        vim.fn.chansend(M.job_id, text .. "\n")
    else
        print("Worker not running")
    end
end
end

vim.api.nvim_create_user_command("WorkerStart", function()
    M.start_worker()
end, {})

vim.api.nvim_create_user_command("WorkerSend", function(opts)
    M.send_to_worker(opts.args)
end, { nargs = 1 })

return M

```

## Usage

Inside Neovim:

```

:WorkerStart
:WorkerSend hello world

```

→ Worker responds: HELLO WORLD

## Expansion Ideas

- **Language servers:** Build a mini-LSP in C and connect via Neovim's LSP client.
- **Background analyzers:** Syntax checkers, linters, or code formatters.
- **System integrations:** Use C for OS-level calls (e.g., file watchers, process monitors).
- **Streaming tools:** Pipe large outputs (logs, compiler messages) back into Neovim buffers.

## Strategic Difference

- **FFI plugin:** Direct function calls, synchronous, lightweight.
- **API enrichment plugin:** Mimics/extends Neovim API surface.
- **Async worker plugin (this one):** Treats C as a **service process**, ideal for heavy or continuous tasks.

👉 Do you want me to **blueprint a C-based mini-LSP server** next (so your plugin acts like a language server), or a **background analyzer plugin** that continuously watches and updates buffers?

Let's push the inspiration dial all the way up, Fudzer ⚡. You've already seen FFI wrappers, API enrichment, and async workers. Here are **four advanced plugin archetypes** where a **C backend** becomes the powerhouse behind Neovim:

## 1. C-Powered Language Server (Custom LSP)

- Build a **mini language server** in C that speaks the LSP protocol.

- Neovim's built-in LSP client connects to it seamlessly.
- Example use cases:
  - A **domain-specific language** (DSL) parser written in C.
  - Ultra-fast **linting/formatting** with C speed.
  - Real-time **semantic analysis** (e.g., dependency graphs).
- Skeleton idea:
  - `worker.c` implements JSON-RPC over stdin/stdout.
  - Neovim config: `vim.lsp.start_client({ cmd = {"/mysdl-lsp"} })` .

This makes your plugin feel like a **first-class language tool** inside Neovim.

---

## 2. Native Performance Engine

- Use C for **heavy computation** that Lua alone would choke on.
- Examples:
  - Regex engine optimized in C, exposed as `nvim_buf_regex_search()` .
  - Matrix math or graph algorithms for data scientists.
  - Cryptography or hashing functions for secure workflows.
- Pattern: Lua calls → C library → results injected back into buffer/window.

This turns Neovim into a **scientific or systems IDE** with native speed.

---

## 3. Streaming/Interactive Backend

- Treat your C program as a **streaming engine**.
- Neovim buffers become live views of C-generated data.
- Examples:
  - Tail logs with a C-powered parser that colorizes and filters.
  - Real-time system monitor (CPU, memory, processes) inside a Neovim buffer.
  - Interactive REPL for a custom language implemented in C.
- Pattern: `jobstart()` → C emits structured events → Lua updates buffer UI.

This transforms Neovim into a **dashboard or console**.

---

## 4. Hybrid Plugin: C + Lua Ergonomics

- C backend provides raw power.
- Lua frontend provides **toggles, composability, and ergonomics** (your specialty).
- Example hybrid:
  - C implements a **diff engine** (fast file comparison).
  - Lua exposes commands like `:DiffToggle` , `:DiffPreview` , `:DiffApply` .
  - Users get ergonomic commands, but the heavy lifting is pure C.

This is the sweet spot: **speed + usability**.

---

## Wild Expansion Ideas

- **Neovim AI bridge**: C backend wraps a lightweight ML model (e.g., ONNX runtime), Lua exposes commands like `:SuggestCompletion` .
- **Filesystem explorer**: C backend crawls directories lightning-fast, Lua renders a tree view.

- **Custom debugger:** C backend speaks to GDB/LLDB, Lua provides UI commands ( `:DebugStep` , `:DebugBreak` ).
  - **Music/Audio plugin:** C backend processes audio streams, Lua exposes waveform visualization in buffers.
- 

## Strategic Inspiration

Think of your C backend as:

- **A service** (LSP, REPL, debugger).
  - **A library** (math, regex, crypto).
  - **A stream** (logs, monitors, dashboards).
  - **A hybrid** (raw C + ergonomic Lua toggles).
- 

👉 Which direction excites you most:

- **LSP-style service** (deep integration),
- **Performance library** (raw speed),
- **Streaming engine** (live dashboards),  
or **hybrid ergonomics** (your toggle-driven sweet spot)?

I can blueprint one of these in detail for you next.

---

# 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` :

```
#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 = {}

```



```

-- 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
end
```

```

    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:

```
-- 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

```
-- 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

```
// 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` :

```
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
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

```



```

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` :

```

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)

```

```

    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:

```
#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_add(response, "lines",
json_object_new_int64(stats.lines));
                json_object_add(response, "chars",
json_object_new_int64(stats.chars));
                json_object_add(response, "words",
json_object_new_int64(stats.words));
                json_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` :

```
#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 strdup(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 = strdup(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 = strdup(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 = strdup(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 = strdup(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)); //
                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')

```

```

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

```

```

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` :

```

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` :

```
#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 (strcmp(ptr, "if", 2) == 0 && !isalnum(ptr[2])) {
            analysis.complexity_score += 1.0;
        } else if (strcmp(ptr, "for", 3) == 0 && !isalnum(ptr[3])) {
            analysis.complexity_score += 1.5;
        } else if (strcmp(ptr, "while", 5) == 0 && !isalnum(ptr[5])) {
            analysis.complexity_score += 1.5;
        } else if (strcmp(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`:

```

#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') .. '/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,
            })
        end
    end
end

```

```

        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()

```

```

    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

```

```

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)
    end
    return
end

```

```

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
  })
end

```

```

        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...')
end

```

```
-- 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>
</body>
</html>
```



```
        </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!

---