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:

```c
#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`

```lua
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**.
