# 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 `rplugin/` manifest.
- Example manifest ( `rplugin.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`

```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`

```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`

```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`

```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)
```

```lua
  if M.job_id then
    vim.fn.chansend(M.job_id, text .. "\n")
  else
    print("Worker not running")
  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/extents 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 = {"./mydsl-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.

---