# 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

```c
#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 }
        }
        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)

```c
#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)

```lua
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

    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)**

```lua
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

    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 -lncurses
```

### 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)

```c
#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)**

```c
#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)

```lua
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)

```c
#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?

---