

How to quickly launch local web servers for rapid testing purposes using mshell.

1. Start gui editor with command “edi”
2. Put requests to chosen LLM inside editor:

```
ollama1exec "Write a small web server in bash for port 8085 and run it in the background so that it responds with 'Bash is here' and print version of Bash to any request. Everything should be in English. No additional comments - only the code."
```

```
ollama1exec "Write a small web server in C for port 8086 that runs in the background and responds to requests with 'C Language is here' and print version of C. Everything should be in English. No additional comments - only the code."
```

```
ollama1exec "Write a small C++ web server for port 8087 that runs in the background and responds to requests with 'C++ is here'. All in English. No additional comments - only the code."
```

```
ollama1exec "Write a small web server in Go for port 8089 and run it in the background so that it responds with 'Go is here' and print version of Go to
```

any request. Everything in English. No additional comments - only the code."

**ollama1exec** "Rust server 8088. Get rustc and cargo versions trim output store as strings. Create body variable with Rust is here newline rustc version newline cargo version. Create HTTP response with Content-Length header using body length. Write response and flush. TcpListener loop. Code only."

**ollama1exec** "Write a small web server in Python for port 8090 and run it in the background so that it responds with 'Python is here' and print version of Python to any request. Everything should be in English. No additional comments - only the code."

**ollama1exec** "Write a small web server in Lua for port 8091 that runs in the background and responds to requests with 'Lua is here' and print version of Lua. All in English. No additional comments - only the code."

Session and generated code is here:

```
/home/igor > edi
```

```
GUI Editor opened (PID 15376)
```

```
You can continue using mshell while editing
```

```
/home/igor > \U0001f680 Starting mshell GUI Editor
```

\U0001f4cd Pipe: /tmp/mshell\_pipe\_15375

Sent to mshell (1660 bytes)

Received from GUI editor:

-----

ollama1exec "Write a small web server in bash for port 8085 and run it in the background so that it responds with 'Bash is here' and print version of Bash to any request. Everything should be in English. No additional comments - only the code."

ollama1exec "Write a small web server in C for port 8086 that runs in the background and responds to requests with 'C Language is here' and print version of C. Everything should be in English. No additional comments - only the code."

ollama1exec "Write a small C++ web server for port 8087 that runs in the background and responds to requests with 'C++ is here'. All in English. No additional comments - only the code."

ollama1exec "Write a small web server in Go for port 8089 and run it in the background so that it responds with 'Go is here' and print version of Go to any request. Everything in English. No additional comments - only the code."

ollama1exec "Rust server 8088. Get rustc and cargo versions trim output store as strings. Create body variable with Rust is here  
newline rustc version newline cargo version. Create HTTP response

with Content-Length header using body length. Write response and flush. TcpListener loop. Code only."

ollama1exec "Write a small web server in Python for port 8090 and run it in the background so that it responds with 'Python is here' and print version of Python to any request. Everything should be in English. No additional comments - only the code."

ollama1exec "Write a small web server in Lua for port 8091 that runs in the background and responds to requests with 'Lua is here' and print version of Lua. All in English. No additional comments - only the code."

```
-----  
  
```bash  
#!/bin/bash  
  
while true; do  
    echo -e "HTTP/1.1 200 OK\r\nContent-Type:  
text/plain\r\nConnection: close\r\n\r\nBash is here\r\nBash version:  
$BASH_VERSION" | nc -l -p 8085 -q 1  
done &  
...  
```c  
  
#include <stdio.h>  
  
#include <stdlib.h>
```

```
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>

int main() {
    if(fork() != 0) exit(0);

    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char response[512];

    sprintf(response, "HTTP/1.1 200 OK\r\nContent-Type:
text/plain\r\nConnection: close\r\n\r\nC Language is here\r\nC
Standard: %ld", __STDC_VERSION__);

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(8086);

    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
```

```
listen(server_fd, 3);

signal(SIGCHLD, SIG_IGN);

while(1) {
    new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen);
    if(fork() == 0) {
        char buffer[1024] = {0};
        read(new_socket, buffer, 1024);
        write(new_socket, response, strlen(response));
        close(new_socket);
        exit(0);
    }
    close(new_socket);
}
return 0;
}
```
  


```
```cpp
#include <iostream>
#include <string>
#include <cstring>
#include <unistd.h>
```


```

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>

int main() {
    if(fork() != 0) exit(0);

    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    std::string response = "HTTP/1.1 200 OK\r\nContent-Type:
text/plain\r\nConnection: close\r\n\r\nC++ is here\r\nC++ Standard: " +
std::to_string(__cplusplus);

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(8087);

    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    listen(server_fd, 3);

    signal(SIGCHLD, SIG_IGN);
```

```

while(true) {
    new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen);
    if(fork() == 0) {
        char buffer[1024] = {0};
        read(new_socket, buffer, 1024);
        write(new_socket, response.c_str(), response.length());
        close(new_socket);
        exit(0);
    }
    close(new_socket);
}
return 0;
}
...
```go
package main

import (
    "fmt"
    "net/http"
    "runtime"
)

```

```
func handler(w http.ResponseWriter, r *http.Request) {  
    fmt.Fprintf(w, "Go is here\nGo version: %s", runtime.Version())  
}
```

```
func main() {  
    http.HandleFunc("/", handler)  
    http.ListenAndServe(":8089", nil)  
}
```

```
```
```

```
```rust
```

```
use std::net::{TcpListener, TcpStream};
```

```
use std::io::prelude::*;
```

```
use std::process::Command;
```

```
fn handle_client(mut stream: TcpStream) {
```

```
    let rustc_version =
```

```
String::from_utf8_lossy(&Command::new("rustc").arg("--  
version").output().unwrap().stdout).trim().to_string();
```

```
    let cargo_version =
```

```
String::from_utf8_lossy(&Command::new("cargo").arg("--  
version").output().unwrap().stdout).trim().to_string();
```

```
    let body = format!("Rust is here\n{}\n{}", rustc_version,  
cargo_version);
```

```
    let response = format!("HTTP/1.1 200 OK\r\nContent-Type:  
text/plain\r\nContent-Length: {}\r\nConnection: close\r\n\r\n{}",  
body.len(), body);
```

```
    stream.write_all(response.as_bytes()).unwrap();  
    stream.flush().unwrap();  
}
```

```
fn main() {
```

```
    let listener = TcpListener::bind("127.0.0.1:8088").unwrap();
```

```
    for stream in listener.incoming() {
```

```
        match stream {
```

```
            Ok(stream) => {
```

```
                std::thread::spawn(|| {
```

```
                    handle_client(stream);
```

```
                });
```

```
            }
```

```
            Err(_) => {}
```

```
        }
```

```
    }
```

```
}
```

```
```
```

```
```python
```

```
import http.server
import socketserver
import sys
import os

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/plain')
        self.end_headers()
        response = f"Python is here\nPython version: {sys.version}"
        self.wfile.write(response.encode())

if __name__ == "__main__":
    if os.fork() == 0:
        with socketserver.TCPServer(("", 8090), MyHandler) as httpd:
            httpd.serve_forever()
...
```lua
local socket = require("socket")

if os.execute("echo $$ > /dev/null &") then
    local server = socket.bind("*", 8091)
```

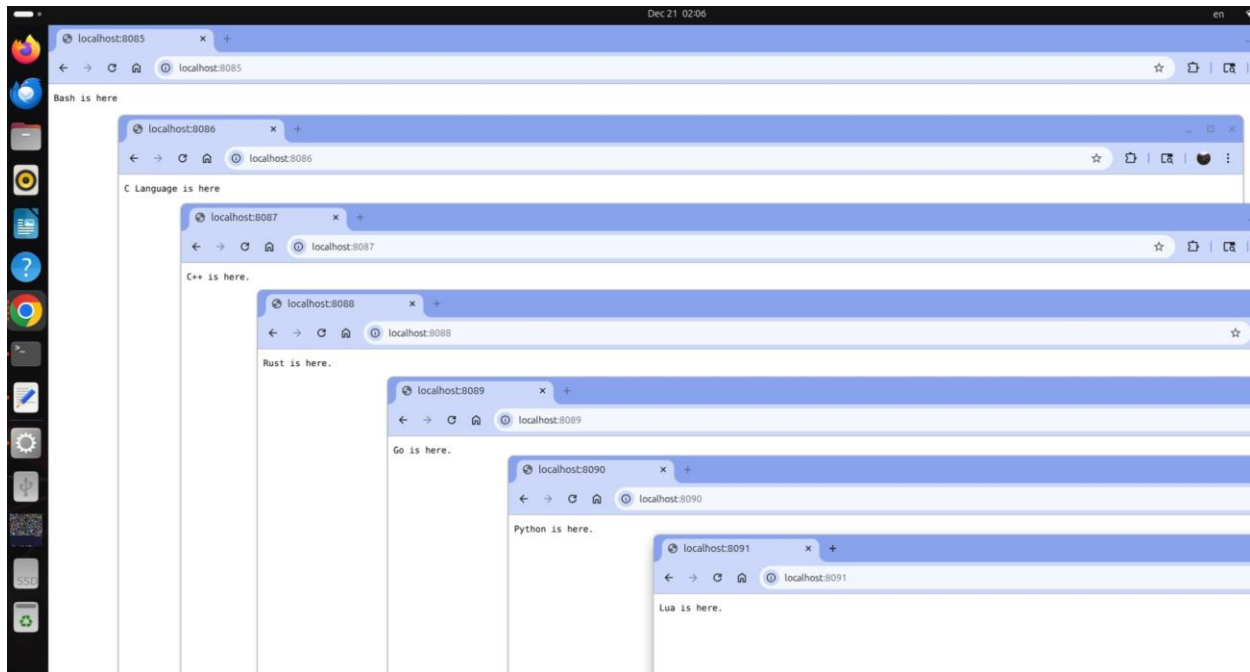
```
while true do
    local client = server:accept()
    local request = client:receive()

    local response = string.format("HTTP/1.1 200 OK\r\nContent-
Type: text/plain\r\nConnection: close\r\n\r\nLua is here\r\nLua version:
%s", _VERSION)

    client:send(response)
    client:close()
end
end
...

```

Testing web-servers on different languages will start automatically. You can start so many instances of each web-server that you want, just change prompt to LLM.



In this demo, upon request to mshell, web servers written in Bash, C, C++, Go, Rust, Python, and Lua are created and launched. The code and the number of instances depend on your prompt when querying the LLM.

You can obtain the generated code for different languages in the Note dedicated to this topic.

#### 1. Line and Character Counts:

Language/Lines/Characters (incl. \n)

Bash/10/175

C/41/1161

C++/39/1150

Go/16/279

Rust/29/1054

Python/17/516

Lua/15/459

## 2. Short Analysis by Language

Bash (10 lines, 175 chars)

The shortest both in lines and characters. It delegates almost everything to nc, so the script remains minimal.

C (41 / 1161)

One of the heaviest. Manual socket handling, low-level APIs, forking, and HTTP response building inflate the size significantly.

C++ (39 / 1150)

Similar to C in structure and size because it relies on POSIX sockets; the use of `std::string` and `std::to_string` adds minimal overhead.

Go (16 / 279)

Very compact thanks to Go's standard net/http library. A complete web server in just a few lines.

Rust (29 / 1054)

Comparable to C/C++ in size due to manual socket handling and invoking external commands for version detection; more robust formatting but higher code volume.

Python (17 / 516)

Medium-sized. The standard `http.server` and `socketserver` modules help reduce complexity, but class structure adds a little overhead.

Lua (15 / 459)

Quite compact. Uses LuaSocket to provide a short and clear server implementation.

### 3. Summary Table (Comparative Results)

Category	Language(s)/Notes
----------	-------------------

Shortest overall code/Bash (10 lines / 175 chars)/The minimal implementation using nc.	
--	--

Smallest real HTTP server (library-based)/Go (279 chars)/Shortest among the “full” server implementations.	
--	--

Middle-sized implementations/Lua (459), Python (516)/Library support makes them compact but not minimal.	
--	--

Largest and most verbose/C (1161), C++ (1150), Rust (1054)/Manual socket work increases size significantly.	
---	--

P.S.: The example does not aim to prove which language is better or worse (each has its own advantages and disadvantages). What is being displayed is what the LLM model generates and showing the application of mshell for solving practical problems, in this example, rapid testing. Any attempts to engage in discussions about the qualities of languages will be ignored.

I. Lukyanov, Art2Dec Soft Lab.

12.25.2025

---