Pure Lua language Patterns for mshell Workflow — Complete Reference Guide (p1–p24)

Pure Lua language Edition — Art2Dec SoftLab, March 24th 2026

Pure Edition Lua-Only · No Python, C, C++, Rust, Go, or bash — only `Lua`, `mshell` directives, and LLM blocks.

What is mshell?

mshell is a polyglot UNIX shell environment for AI and data workflows. It integrates multiple programming languages and LLM models into a single unified execution pipeline. Lua blocks are executed automatically via `lua5.4`. Variables flow between blocks via named files. LLM directives inject model responses as first-class variables that any subsequent block can consume.

Variable System

Reading a variable:

```lua
local p = os.getenv("MSH_VAR_varname")
local f = io.open(p, "r")
local val = f:read("*l"):gsub("%s+$", "")
f:close()
```

Reading full multi-line variable:

```lua
local f = io.open(os.getenv("MSH_VAR_varname"), "r")
local text = f:read("*all"); f:close()
```

Writing a variable (multiple >outvar block):

```lua
-- stdout is NOT captured when multiple >outvar are declared — write directly
local f = io.open(os.getenv("MSH_VAR_varname"), "w")
f:write(value .. "\n"); f:close()
```

Executing LLM-generated Lua code:

```lua
local fn, err = loadfile(os.getenv("MSH_VAR_code"))
if fn then fn() else print("Error: " .. tostring(err)) end
```

FOREACH list (no trailing newline):

```lua
io.write("item1\nitem2\nitem3")  -- io.write not print — no trailing newline
```

WHILE loop counter update:

```lua
local p = os.getenv("MSH_VAR_counter")
local f = io.open(p, "r"); local val = tonumber(f:read("*l")) or 0; f:close()
local f2 = io.open(p, "w"); f2:write(tostring(val + 1) .. "\n"); f2:close()
```

Critical Rules

1. **`lua5.4` only** — mshell executes Lua blocks with `lua5.4`. All code must be compatible.
2. **`loadfile` for code execution** — Use `loadfile(path)` to run LLM-generated Lua code, not `dofile` or `require`.
3. **Multiple `>outvar`** — Write directly to `MSH_VAR_*` files. Stdout is NOT captured.
4. **WHILE counter** — Read/write `MSH_VAR_*` files directly with `io.open`.
5. **FOREACH list** — Use `io.write` (no trailing newline). One item per line.
6. **TRY/CATCH** — Trigger CATCH with `os.exit(1)`. CATCH block prints literal `"try_block_failed"` — do NOT use `<errvar`.
7. **LLM prompts** — Add `You MUST respond` to prevent empty responses.
8. **Evaluator prompts** — For LOOP patterns, evaluator must reply with ONLY one word (`ACCEPTED` or `REJECTED`). Add: `Do NOT add any explanation — just the single word.`
9. **No `\uXXXX` in Lua strings** — Lua does not support `\u` unicode escapes. Use actual Unicode characters directly in string literals.
10. **Read strings** — Use `:gsub("%%s+$","")` to strip trailing whitespace/newlines from variable values.

Pattern Summary Table

#	Pattern	Nodes	Models	Key Technique
1	Linear Data Pipeline	—	—	7-stage sequential Lua pipeline
2	LLM in the Middle	—	@1	Lua → LLM → Lua
3	Fan-Out	—	@1	One var, multiple consumers
4	LLM Code Gen + Execute	—	@1	loadfile execution
5	Two-LLM Review Chain	—	@1, @2	Generate → Review → Improve

#	Pattern	Nodes	Models	Key Technique
6	Parallel 3-Model Query	—	@1–@3	Sequential LLM directives
7	Evaluator-Optimizer Loop	LOOP	@1, @2	ACCEPTED/REJECTED single word
8	Multi-Stage + Multi-Model	—	@1, @2	Lua stats + LLM analysis
9	Routing	—	@1	if=route:VALUE conditional
10	Full Pipeline	AWAIT	@1–@3	All patterns combined
11	MShell Native AI	—	@1, @2	ollama1/ollama2 inline
12	Async 3 Models + Synthesis	AWAIT	@1–@3	Parallel async + barrier
13	WHILE Counter	WHILE	@1	io.open write MSH_VAR_status
14	FOREACH List	FOREACH	@1	io.write no trailing newline
15	TRY/CATCH	TRY/CATCH	—	os.exit(1) triggers CATCH
16	SPLIT + MERGE	SPLIT, MERGE	@1, @2	Async parallel + synthesis
17	CONFIG Pipeline	CONFIG	@1, @2	CONFIG docs, lua sets values
18	FOREACH + Async	FOREACH, AWAIT	@1, @2	Parallel per-item LLMs
19	WHILE Quality Gate	WHILE	@1, @2	io.open status on threshold
20	Map-Reduce	SPLIT, MERGE, AWAIT	@1, @2	3 async map + reduce
21	TRY/CATCH + LOOP	TRY/CATCH, LOOP	@1	loadfile, self-correcting
22	Multi-Variable Output	—	@1, @2	io.open write per MSH_VAR_*
23	CONFIG + WHILE + 3M	CONFIG, WHILE	@1–@3	Adaptive quality pipeline
24	FOREACH + TRY/CATCH	FOREACH	@1	string.match fault isolation

Part I — Patterns 1–12: Core Patterns

Pattern 1 — Linear Data Pipeline (Lua)

What it does: Seven sequential Lua stages: raw number → squaring → prime check → factorial → digit sum → FizzBuzz → final report. Pure Lua pipeline, no LLM.

```
print("7")
```

```
local p = os.getenv("MSH_VAR_raw")
local f = io.open(p,"r"); local v = tonumber(f:read("*l")); f:close()
print(v * v)
```

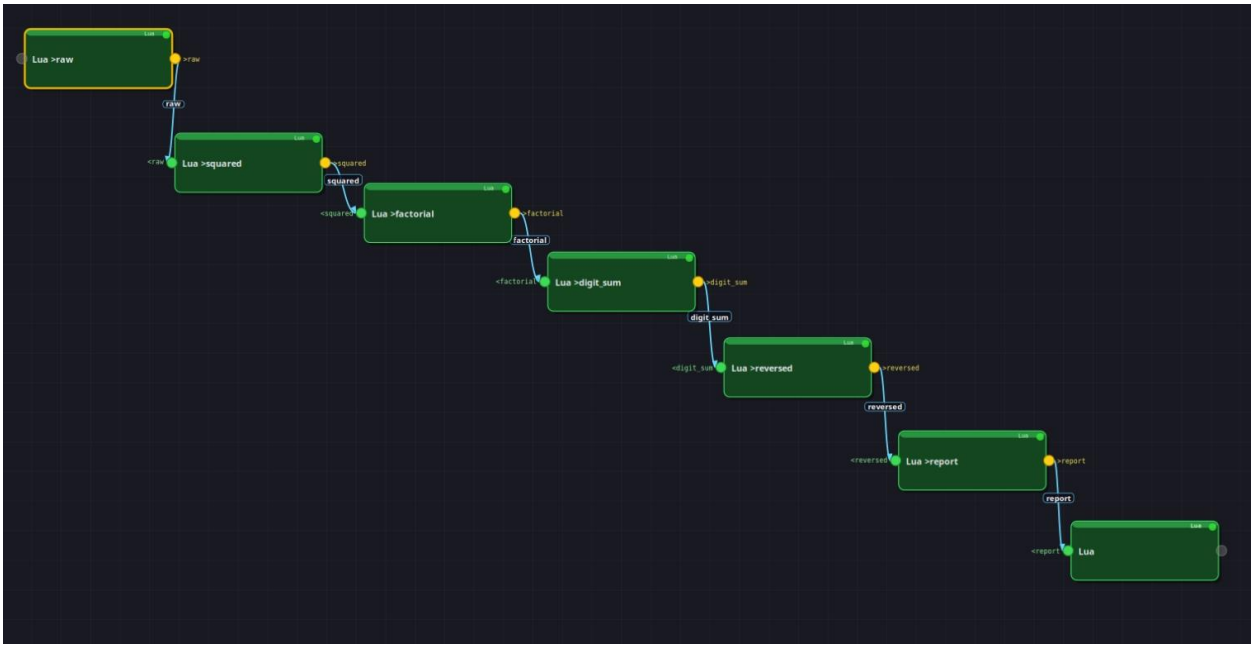
```
local p = os.getenv("MSH_VAR_squared")
local f = io.open(p,"r"); local n = tonumber(f:read("*l")); f:close()
local d = math.floor(n / 10) % 10
local fact = 1
for i = 2, d do fact = fact * i end
print(fact)
```

```
local p = os.getenv("MSH_VAR_factorial")
local f = io.open(p,"r"); local s = f:read("*l"):gsub("%s+",""); f:close()
local total = 0
for c in s:gmatch("%d") do total = total + tonumber(c) end
print(total)
```

```
local p = os.getenv("MSH_VAR_digit_sum")
local f = io.open(p,"r"); local s = f:read("*l"):gsub("%s+",""); f:close()
print(s:reverse())
```

```
local p = os.getenv("MSH_VAR_reversed")
local f = io.open(p,"r"); local v = f:read("*l"):gsub("%s+",""); f:close()
print("Pipeline complete. Final value: " .. v)
```

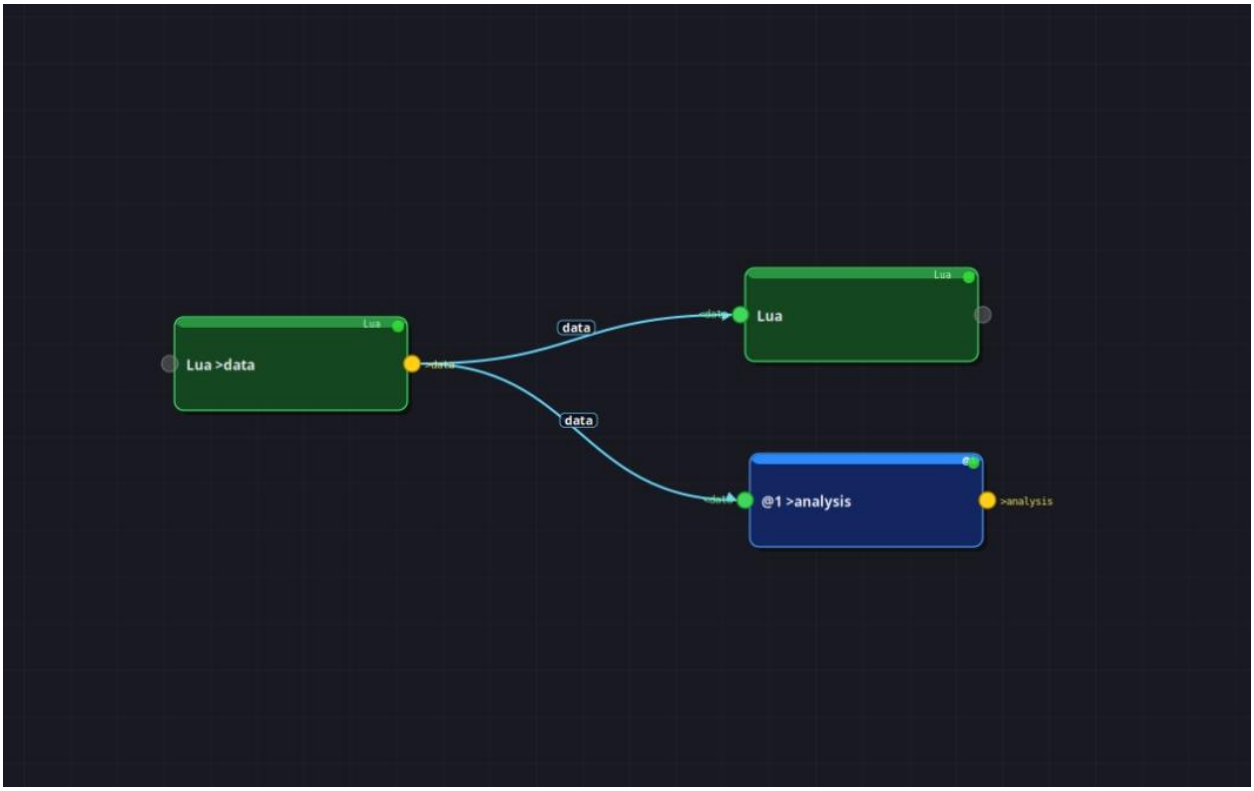
```
local p = os.getenv("MSH_VAR_report")
local f = io.open(p,"r")
print("=== Pipeline Result ===")
print(f:read("*all"):gsub("%s+$",""))
f:close()
```



Pattern 2 — LLM in the Middle

What it does: Lua generates a city temperature table. LLM @1 writes a weather summary. A second Lua block counts words and sentences in the response.

```
print("City temperatures (°C): Oslo=-4, Reykjavik=-1, London=9, Rome=18, Cair
o=28, Bangkok=34")

io.write(io.open(os.getenv("MSH_VAR_data"),"r"):read("*all"))

local p = os.getenv("MSH_VAR_analysis")
local f = io.open(p,"r"); local text = f:read("*all"):gsub("%s+$",""); f:clos
e()
local words = 0
for _ in text:gmatch("%S+") do words = words + 1 end
local sents = 0
for _ in text:gmatch("[%.!%?]") do sents = sents + 1 end
print(string.format("LLM response: %d words, ~%d sentence(s)", words, sents))
print("--- Analysis ---")
print(text)
```



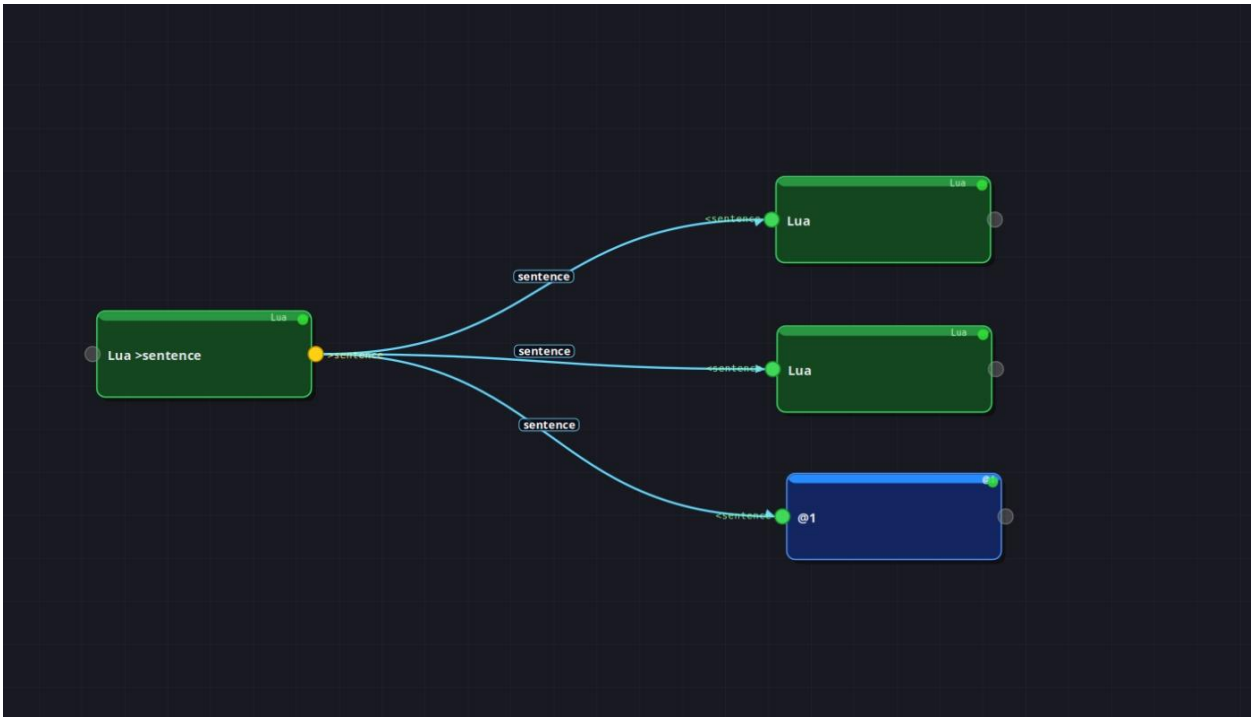
Pattern 3 — Fan-Out: One Variable → Many Consumers

What it does: One Lua block generates a sentence. Three independent consumers read the same variable: Lua counts vowels, Lua reverses words, and LLM @1 identifies mood.

```
print("The wandering moon crossed the silent silver river at midnight")
```

```
local p = os.getenv("MSH_VAR_sentence")
local f = io.open(p,"r"); local s = f:read("*l"):gsub("%s+$",""); f:close()
local vowels = 0
for _ in s:lower():gmatch("[aeiou]") do vowels = vowels + 1 end
print("Lua consumer 1 — vowel count: " .. vowels)
```

```
local p = os.getenv("MSH_VAR_sentence")
local f = io.open(p,"r"); local s = f:read("*l"):gsub("%s+$",""); f:close()
local words = {}
for w in s:gmatch("%S+") do table.insert(words, 1, w) end
print("Lua consumer 2 — reversed words: " .. table.concat(words, " "))
```

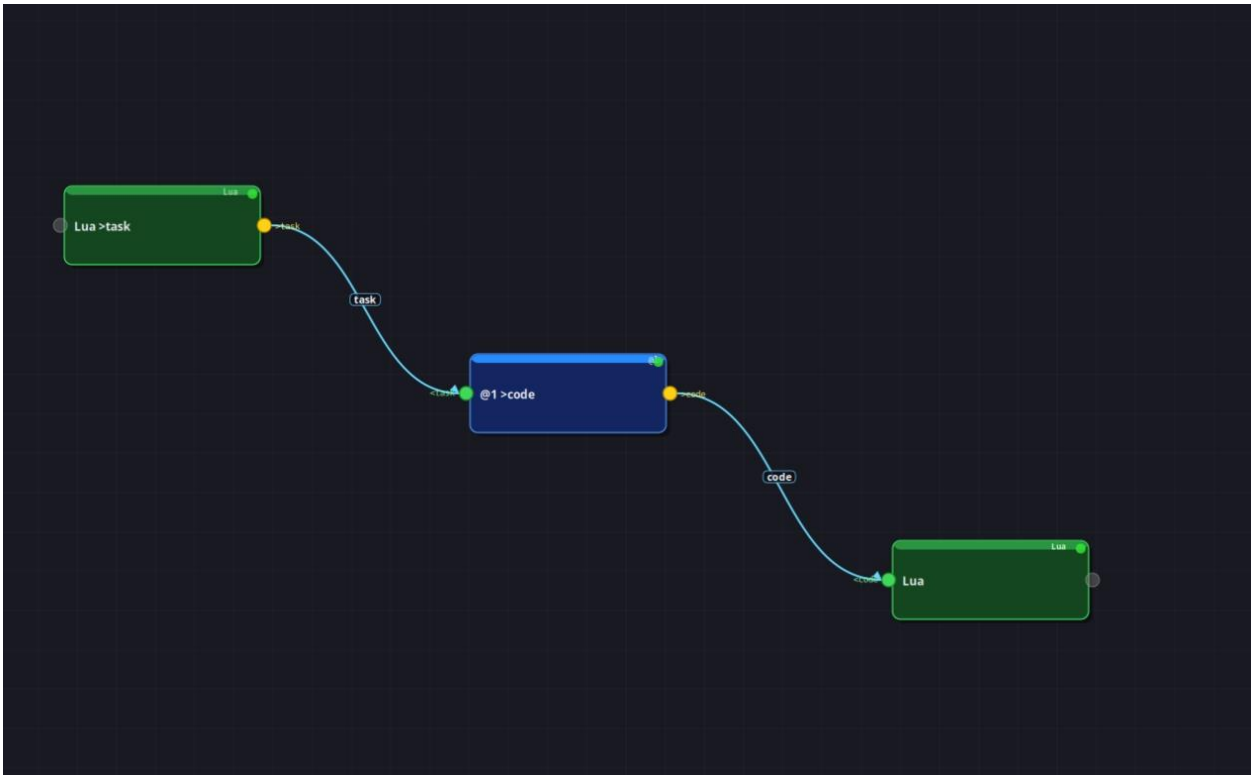


Pattern 4 — LLM Code Generation → Execute via Variable

What it does: Lua writes a task description. LLM @1 generates complete Lua code. A second Lua block reads the generated path, prints the source, then executes it with `loadfile`.

```
print("Write a Lua function called primes(n) that returns the first n primes,
call it with n=8, print each result.")
```

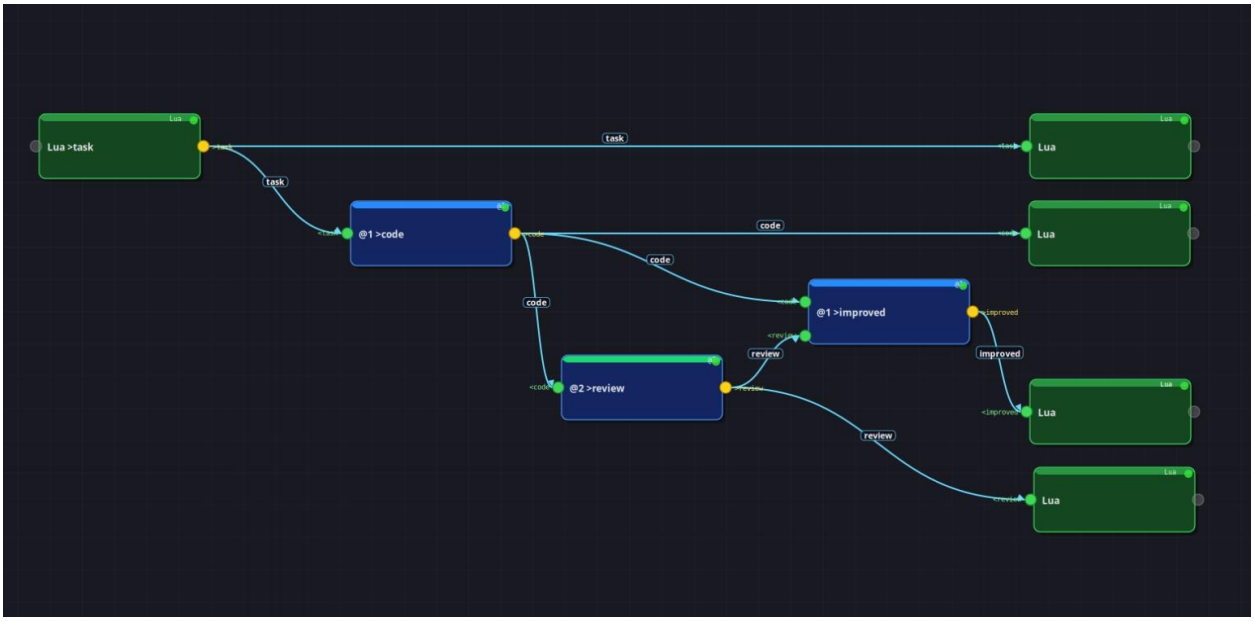
```
local code_path = os.getenv("MSH_VAR_code")
local cf = io.open(code_path,"r"); local src = cf:read("*all"); cf:close()
print("=== Generated Lua code ==="); print(src:gsub("%s+$",""))
print("=== Executing ===")
local fn, err = loadfile(code_path)
if fn then fn() else print("Load error: " .. tostring(err)) end
```



Pattern 5 — Two-LLM Review Chain

What it does: LLM @1 generates a Lua function. LLM @2 reviews it. LLM @1 improves based on the review. A final Lua block executes the improved code with `loadfile`.

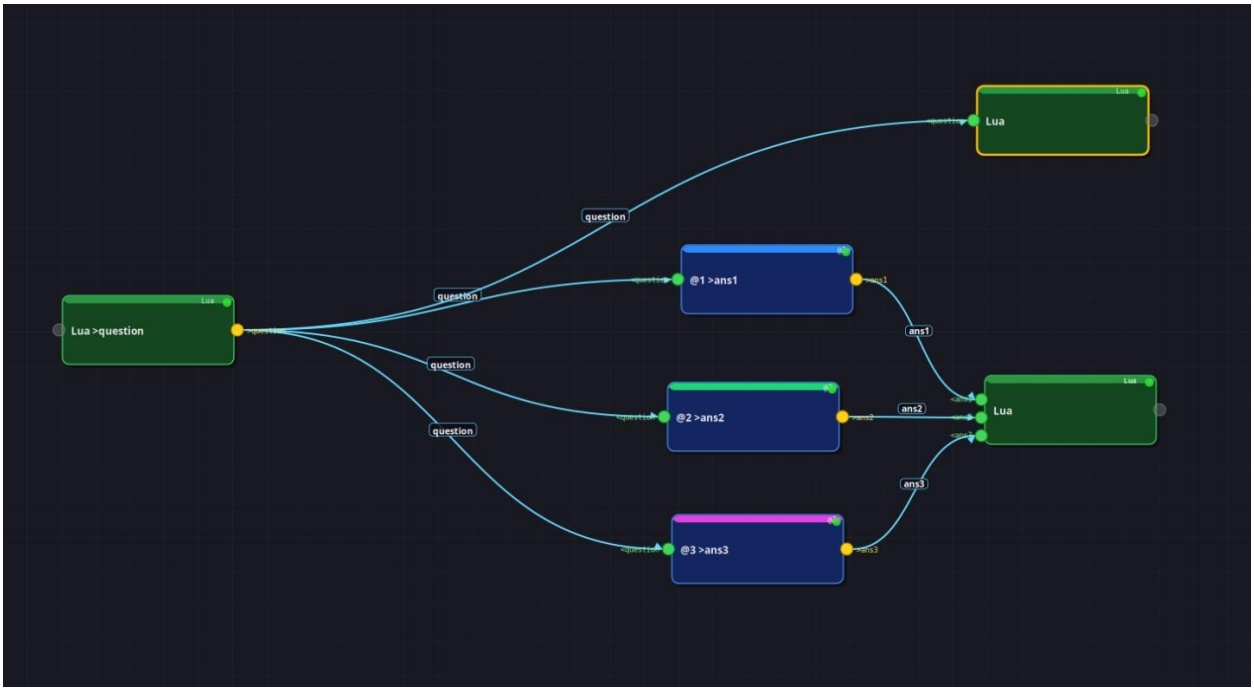
```
print("write a Lua function that computes the nth Fibonacci number using memo
ization")

io.write(io.open(os.getenv("MSH_VAR_task"),"r"):read("*all"))

local p = os.getenv("MSH_VAR_code")
local f = io.open(p,"r"); print("=== Model 1 generated ==="); print(f:read("*
all"):gsub("%s+$","")); f:close()

local p = os.getenv("MSH_VAR_review")
local f = io.open(p,"r"); print("=== Model 2 review ==="); print(f:read("*all
"):gsub("%s+$","")); f:close()

local p = os.getenv("MSH_VAR_improved")
local f = io.open(p,"r"); local src = f:read("*all"); f:close()
print("=== Final improved code ==="); print(src:gsub("%s+$",""))
print("=== Executing ===")
local fn, err = loadfile(p)
if fn then fn() else print("Error: " .. tostring(err)) end
```



Pattern 6 — Parallel 3-Model Query

What it does: Lua formulates a philosophical question. All three LLM models answer independently. Lua formats all three responses in a comparison table.

```
print("If you could remove one cognitive bias from humanity, which would it b
e and why? One sentence.")

io.write(io.open(os.getenv("MSH_VAR_question"),"r"):read("*all"))

for i, v in ipairs({"ans1","ans2","ans3"}) do
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:clo
se()
    print(string.format("=== Model %d (%d chars) ===", i, #t))
    print(t); print()
end
```



Pattern 7 — Evaluator-Optimizer Loop

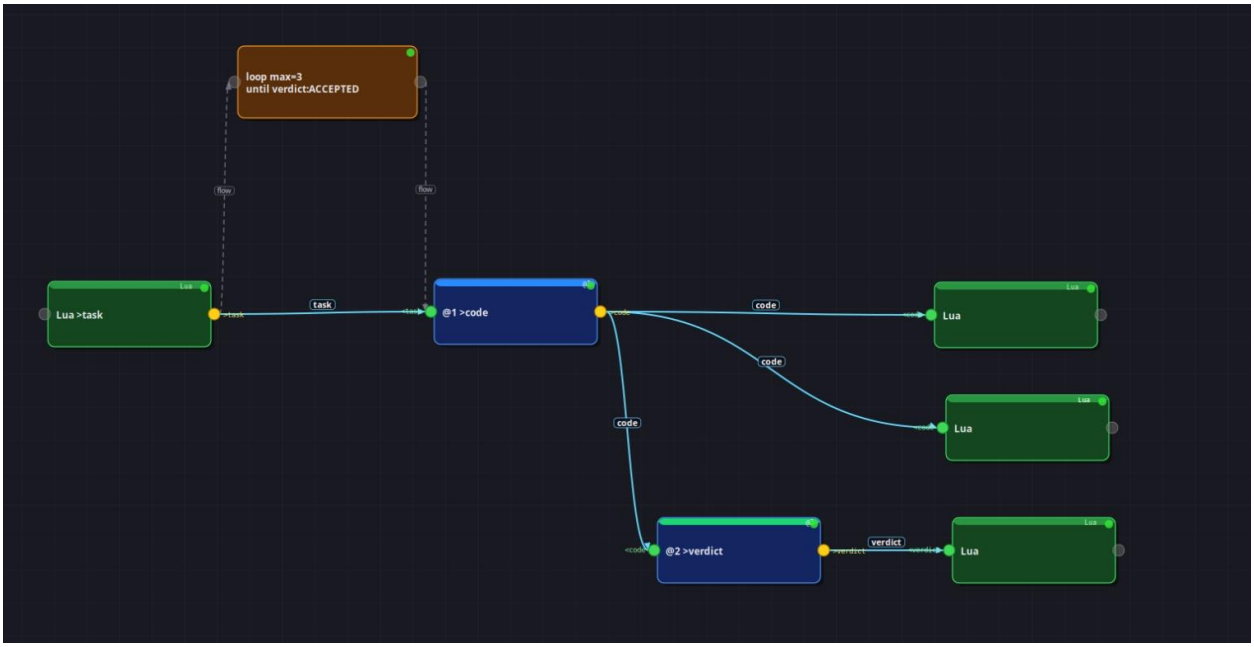
What it does: LLM @1 generates Lua code. LLM @2 evaluates it (ACCEPTED / REJECTED — single word only). Loop repeats up to 3 times. Final Lua block executes the accepted code.

```
print("write a Lua function is_palindrome(s) ignoring spaces and case, test o
n 3 examples")
```

```
local p = os.getenv("MSH_VAR_code")
local f = io.open(p,"r"); print("=== Generated code ==="); print(f:read("*all
"):gsub("%s+$",""")); f:close()
```

```
local p = os.getenv("MSH_VAR_verdict")
local f = io.open(p,"r"); print("=== Verdict ==="); print(f:read("*all"):gsub
("%s+$",""")); f:close()
```

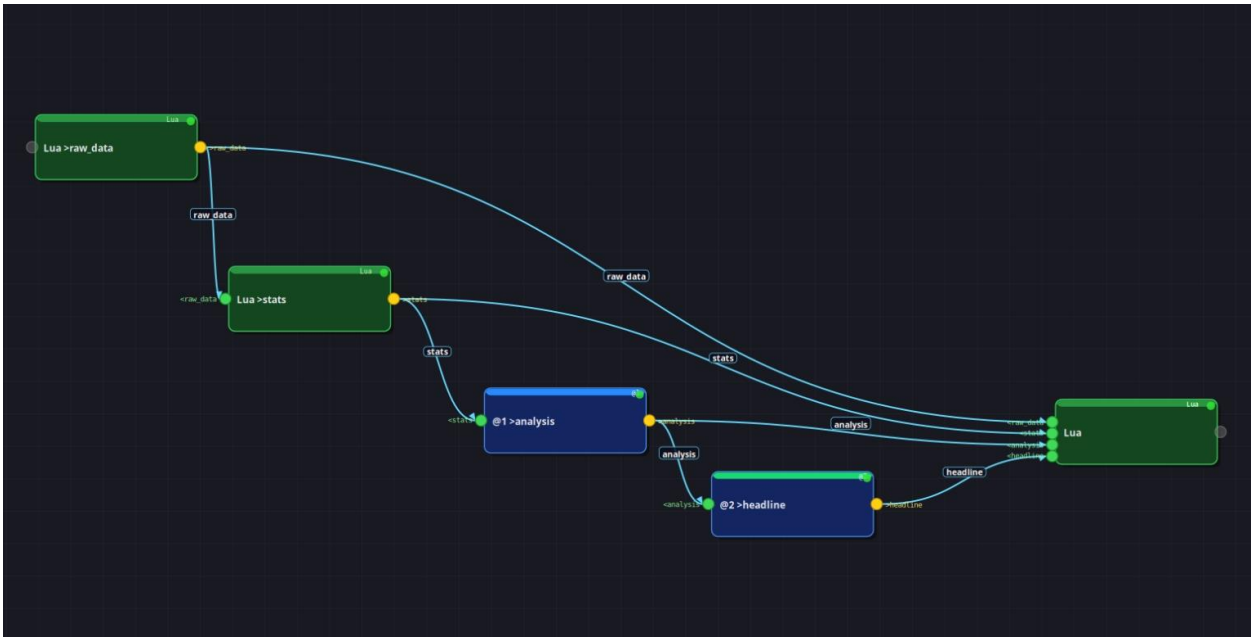
```
local p = os.getenv("MSH_VAR_code")
print("=== Final accepted code ===")
local f = io.open(p,"r"); print(f:read("*all"):gsub("%s+$",""")); f:close()
print("=== Executing ===")
local fn, err = loadfile(p)
if fn then fn() else print("Load error: " .. tostring(err)) end
```



Pattern 8 — Multi-Stage Lua + Multi-Model Pipeline

What it does: Lua computes statistics on a dataset. LLM @1 interprets the data. LLM @2 compresses to a headline. Lua formats a structured report.

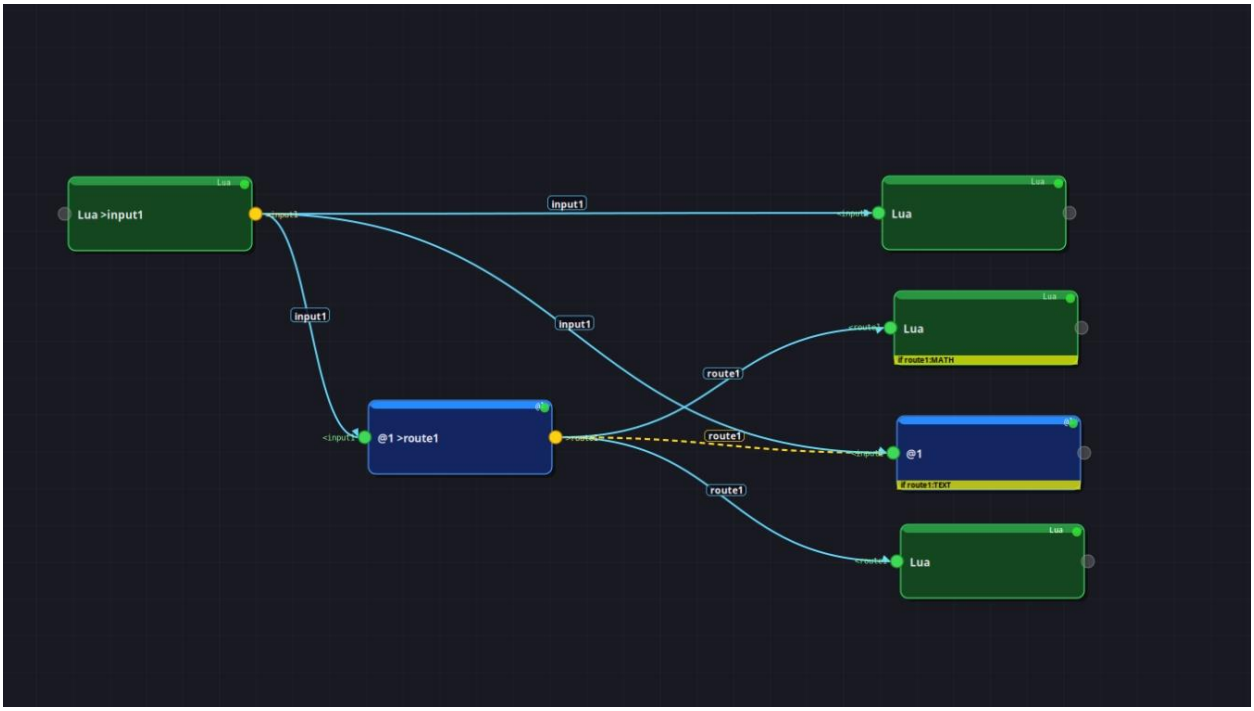
```
math.randomseed(42)
local nums = {}
for i = 1, 10 do table.insert(nums, math.random(0,99)) end
print(table.concat(nums, ","))

local p = os.getenv("MSH_VAR_raw_data")
local f = io.open(p,"r"); local line = f:read("*l"):gsub("%s+$",""); f:close(
)
local nums = {}
for n in line:gmatch("%d+") do table.insert(nums, tonumber(n)) end
local sum = 0; local mn, mx = nums[1], nums[1]
for _, v in ipairs(nums) do
    sum = sum + v
    if v < mn then mn = v end
    if v > mx then mx = v end
end
print(string.format("count=%d mean=%.1f min=%d max=%d sum=%d", #nums, sum/#nu
ms, mn, mx, sum))

for _, v in ipairs({"raw_data","stats","analysis","headline"}) do
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:clo
se()
    print(string.format("[%s]: %s", v, t))
end
```



Pattern 9 — Routing: LLM Classifies → Conditional Lua Branch

What it does: LLM @1 classifies an input into MATH, TEXT, or SORT. Only the matching conditional Lua block runs. Three test cases cover all branches.

```
print("calculate the sum of squares from 1 to 10")

io.write(io.open(os.getenv("MSH_VAR_input1"),"r"):read("*all"))

print("=== MATH branch ===")
local total = 0
for i = 1, 10 do total = total + i*i end
print("Sum of squares 1..10 = " .. total)

local p = os.getenv("MSH_VAR_route1")
local f = io.open(p,"r"); print("Classified as: " .. f:read("*l"):gsub("%s+$"
,"")); f:close()
```

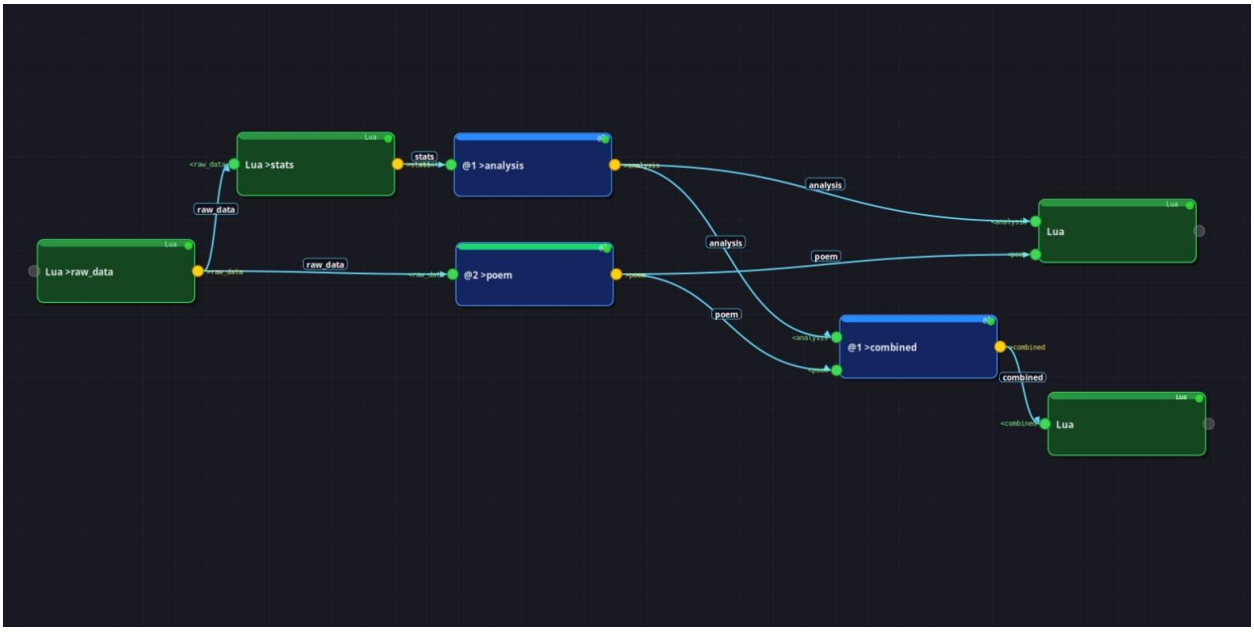


Pattern 10 — Full Pipeline: All Patterns Combined

What it does: Lua computes perfect squares and derives stats. Two LLMs run asynchronously. Await barrier synchronises. LLM @1 synthesises. Lua renders output in a decorative frame.

```
local t = {}
for i = 1, 10 do table.insert(t, i*i) end
print(table.concat(t, " "))

local p = os.getenv("MSH_VAR_raw_data")
local f = io.open(p,"r"); local line = f:read("*l"):gsub("%s+$",""); f:close(
)
local nums = {}
for n in line:gmatch("%d+") do table.insert(nums, tonumber(n)) end
local sum = 0
for _, v in ipairs(nums) do sum = sum + v end
print(string.format("First 10 perfect squares: %s", line))
print(string.format("Sum=%d  Mean=%.1f", sum, sum/#nums))

local function read(v)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:clo
se()
    return t
end
print("=== Analysis ==="); print(read("analysis"))
print(); print("=== Poem ==="); print(read("poem"))
```

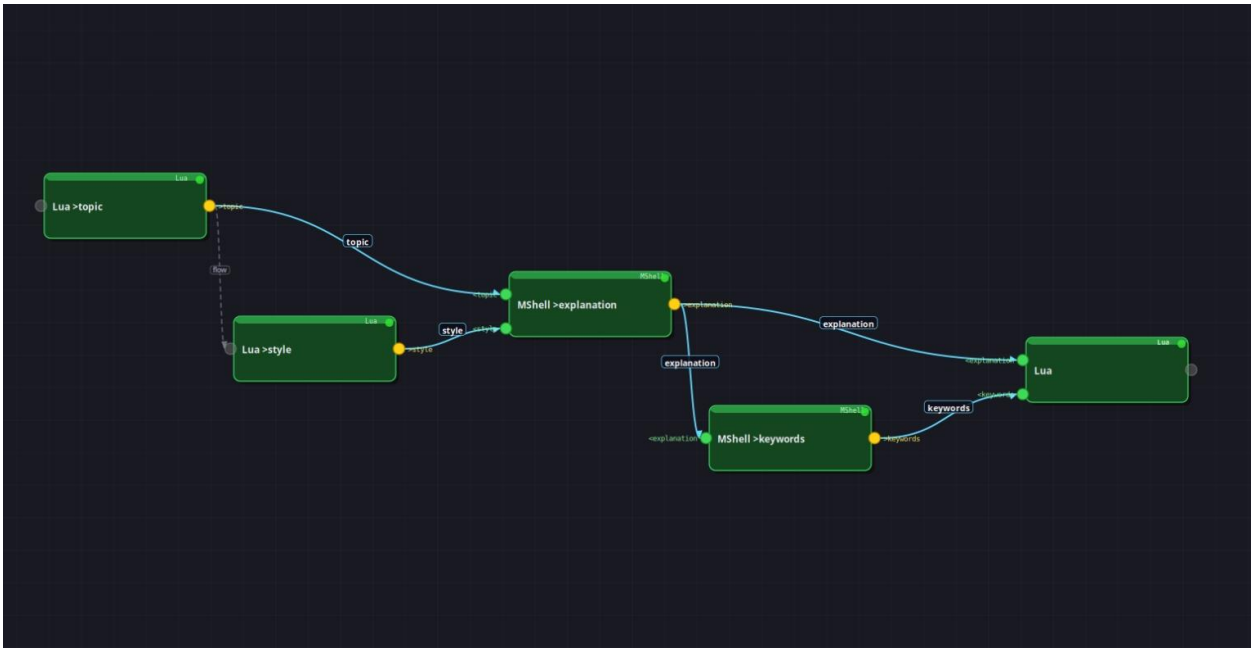
```
local p = os.getenv("MSH_VAR_combined")
local f = io.open(p,"r"); local text = f:read("*all"):gsub("%s+$",""); f:clos
e()
local border = string.rep("=", math.min(#text, 72))
print("=== Final Output ==="); print(border); print(text); print(border)
```



Pattern 11 — MShell Node with Multiple Models

What it does: Lua produces `topic` and `style` variables. Native mshell blocks call `ollama1`/`ollama2` inline. Lua formats the final report.

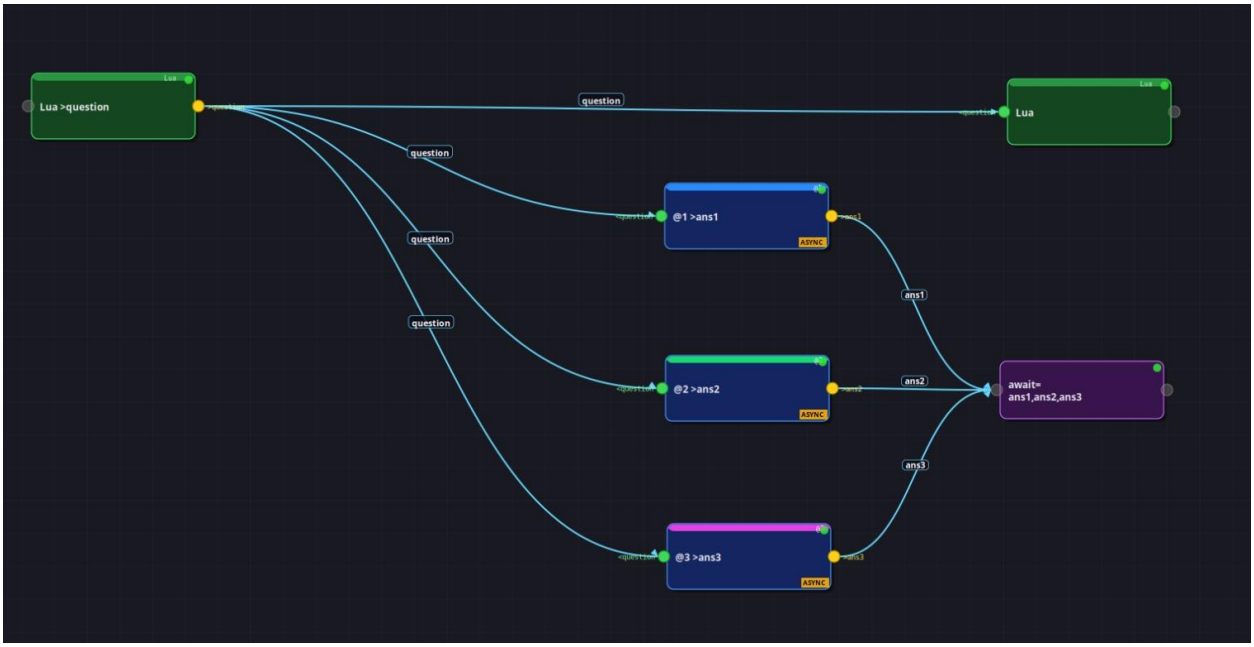
```
print("quantum computing")

print("vivid and analogy-driven")

ollama1 "Explain $topic in a $style way in one sentence"

ollama2 "Extract exactly 3 keywords from: $explanation. Reply with 3 words, c
omma-separated only."

local function read(v)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:clo
se()
    return t
end
print("=== Explanation ==="); print(read("explanation"))
print(); print("=== Keywords ==="); print(read("keywords"))
```



Pattern 12 — Async Parallel 3 Models + Await Barrier + Synthesis

What it does: Three LLMs answer the same question asynchronously with different angles. Await barrier synchronises. LLM @1 synthesises all three into one sentence.

```
print("What are metatables and metamethods in Lua?")

io.write(io.open(os.getenv("MSH_VAR_question"),"r"):read("*all"))



local p = os.getenv("MSH_VAR_final")
local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
print("=== Three perspectives synthesized ==="); print(t)
```



Part II — Patterns 13–24: Advanced Node Types

New node types: **WHILE · FOREACH · TRY/CATCH · SPLIT · MERGE · CONFIG · Multi-Variable Output**

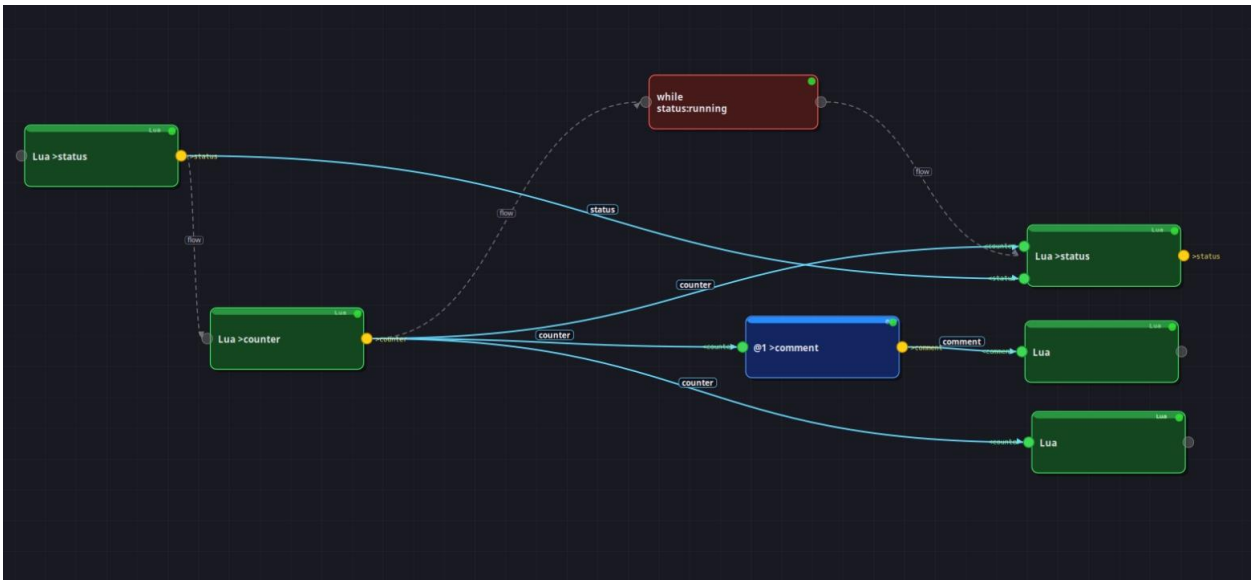
Pattern 13 — WHILE Loop: Iterative Counter with LLM Commentary

What it does: WHILE loop. Each iteration a Lua block increments the counter and writes new status directly to `MSH_VAR_*` files. LLM @1 provides one fact per step. Stops at 4.

```
print("running")

print("0")

local function read(v)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = f:read("*all") or "0"; f:close()
    return t:gsub("%s+","")
end
local function write(v, val)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"w"); f:write(val); f:close()
end
local raw = read("counter")
local val = tonumber(raw:match("%d+") or "0") + 1
write("counter", tostring(val))
write("status", val >= 3 and "done" or "running")
print(val)
```

```
local pc = os.getenv("MSH_VAR_counter")
local fc = io.open(pc,"r"); local cnt = (fc:read("*all") or ""):gsub("%s+","")
); fc:close()
local pm = os.getenv("MSH_VAR_comment")
local fm = io.open(pm,"r"); local msg = fm:read("*all"):gsub("%s+$",""); fm:c
lose()
print(string.format("[iter %s] %s", cnt, msg))
```

```
local p = os.getenv("MSH_VAR_counter")
local f = io.open(p,"r"); local v = (f:read("*all") or ""):gsub("%s+",""); f:
close()
print("=== WHILE done. Final counter = " .. v .. " ===")
```



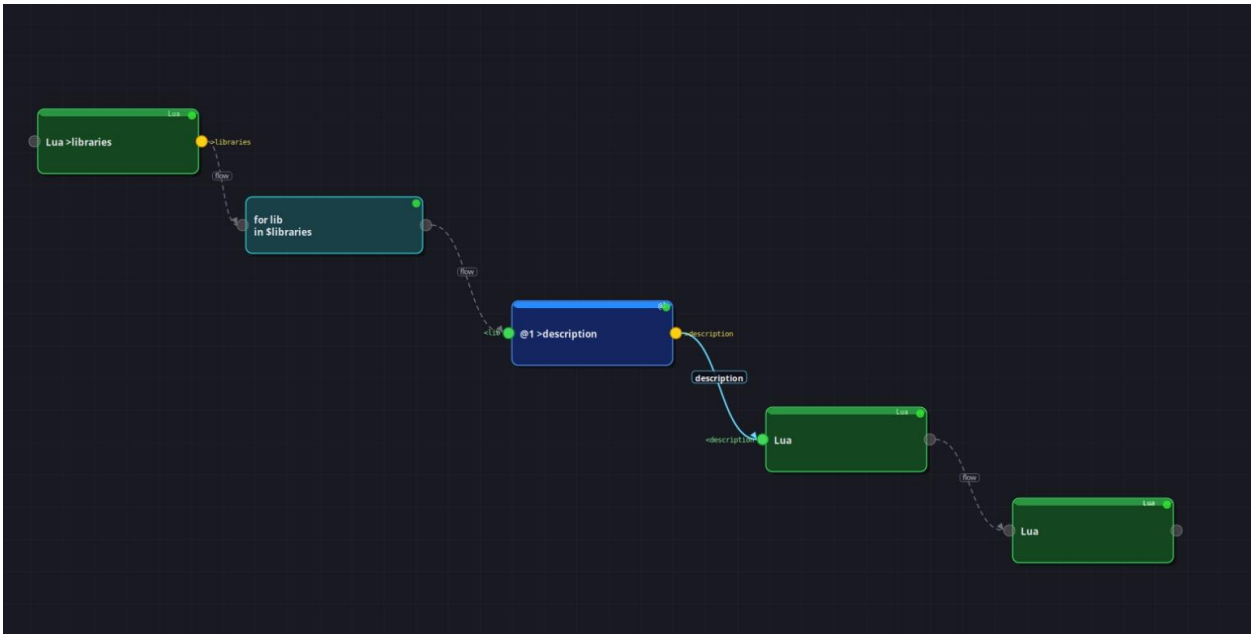
Pattern 14 — FOREACH: LLM Explains Each Lua Library

What it does: Lua creates a newline-separated list of Lua standard libraries. FOREACH iterates. LLM @1 explains each library in one sentence. Lua prints formatted cards.

```
io.write("io\nmath\nstring")
```

```
local pl = os.getenv("MSH_VAR_lib")
local fl = io.open(pl,"r"); local lib = fl:read("*l"):gsub("%s+$",""); fl:clo
se()
local pd = os.getenv("MSH_VAR_description")
local fd = io.open(pd,"r"); local desc = fd:read("*all"):gsub("%s+$",""); fd:
close()
print("--- " .. lib .. " ---"); print(desc); print()
```

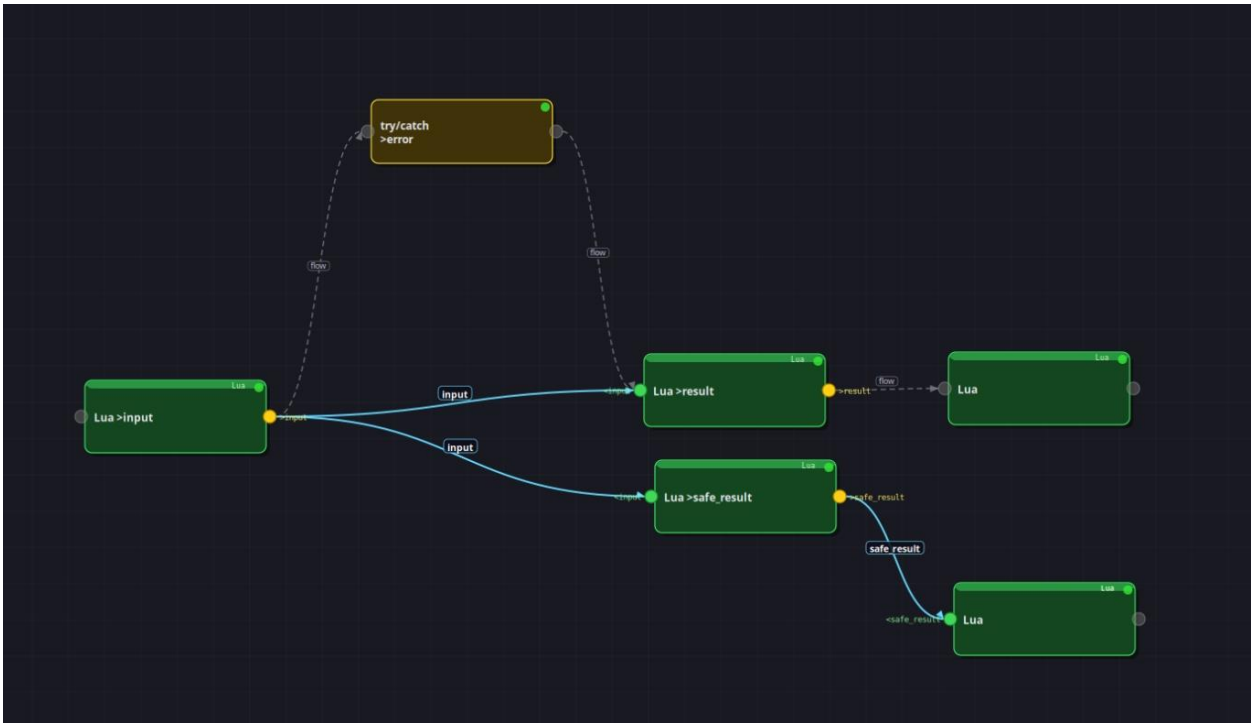
```
print("=== All Lua libraries described ===")
```



Pattern 15 — TRY/CATCH: Safe Execution with Error Capture

What it does: Lua attempts to index a nil value intentionally. `os.exit(1)` triggers CATCH. CATCH prints a hardcoded literal. Safe fallback Lua block counts words instead.

```
print("hello Lua world")

local p = os.getenv("MSH_VAR_input")
local f = io.open(p,"r"); local text = f:read("*l"):gsub("%s+$",""); f:close(
)
local broken = nil
local _ = broken.field   -- intentional nil index error
print(text)
os.exit(1)

print("=== Caught error: try_block_failed ===")
print("Pipeline continues safely.")

local p = os.getenv("MSH_VAR_input")
local f = io.open(p,"r"); local text = f:read("*l"):gsub("%s+$",""); f:close(
)
local count = 0
for _ in text:gmatch("%S+") do count = count + 1 end
print("Safe fallback: word count = " .. count)

local p = os.getenv("MSH_VAR_safe_result")
local f = io.open(p,"r"); print("=== Safe result ==="); print((f:read("*all")
:gsub("%s+$","")))); f:close()
```

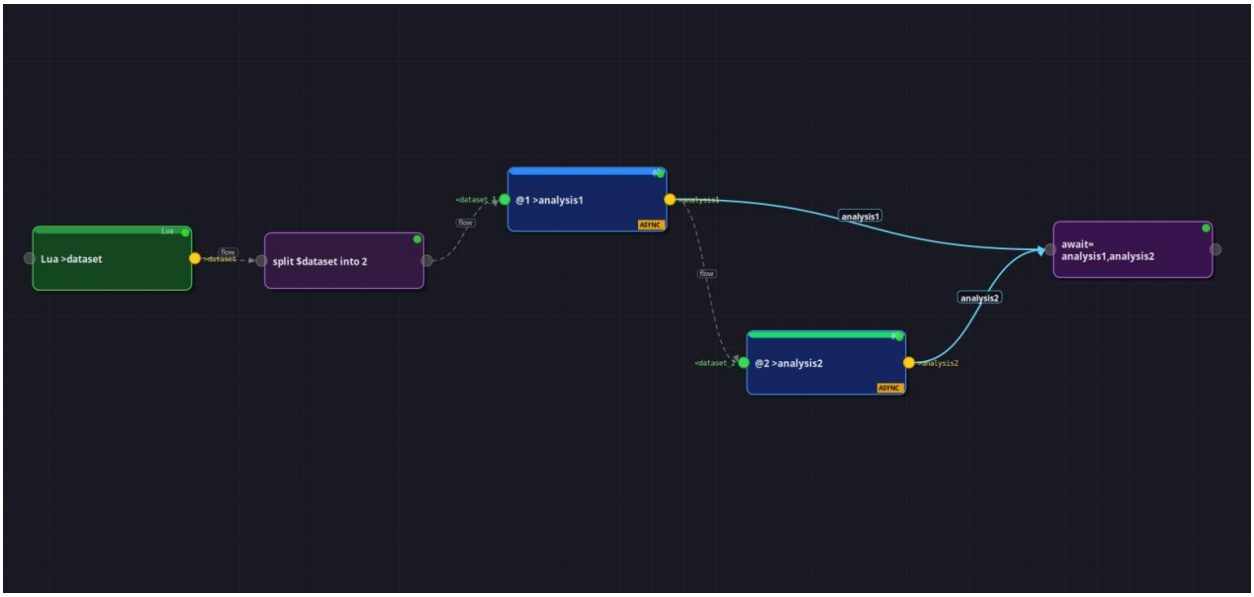


Pattern 16 — SPLIT + MERGE: Divide-and-Conquer Analysis

What it does: Lua creates a two-stock dataset. SPLIT divides by lines. Two async LLMs analyse each stock. Await synchronises. LLM @1 synthesises a comparison.

```
io.write("AAPL:182,179,185,188,191\nMSFT:374,371,378,382,385")
```

```
local function read(v)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:clo
se()
    return t
end
print("=== Part 1: " .. read("dataset_1") .. " ==="); print("Analysis: " .. r
ead("analysis1")); print()
print("=== Part 2: " .. read("dataset_2") .. " ==="); print("Analysis: " .. r
ead("analysis2")); print()
print("=== Merged ==="); print(read("combined"))
```



Pattern 17 — CONFIG Node: Parameterized Pipeline

What it does: CONFIG block documents parameters. Lua blocks set runtime values. LLM @1 explains a topic. LLM @2 extracts keywords. Lua formats a complete report.

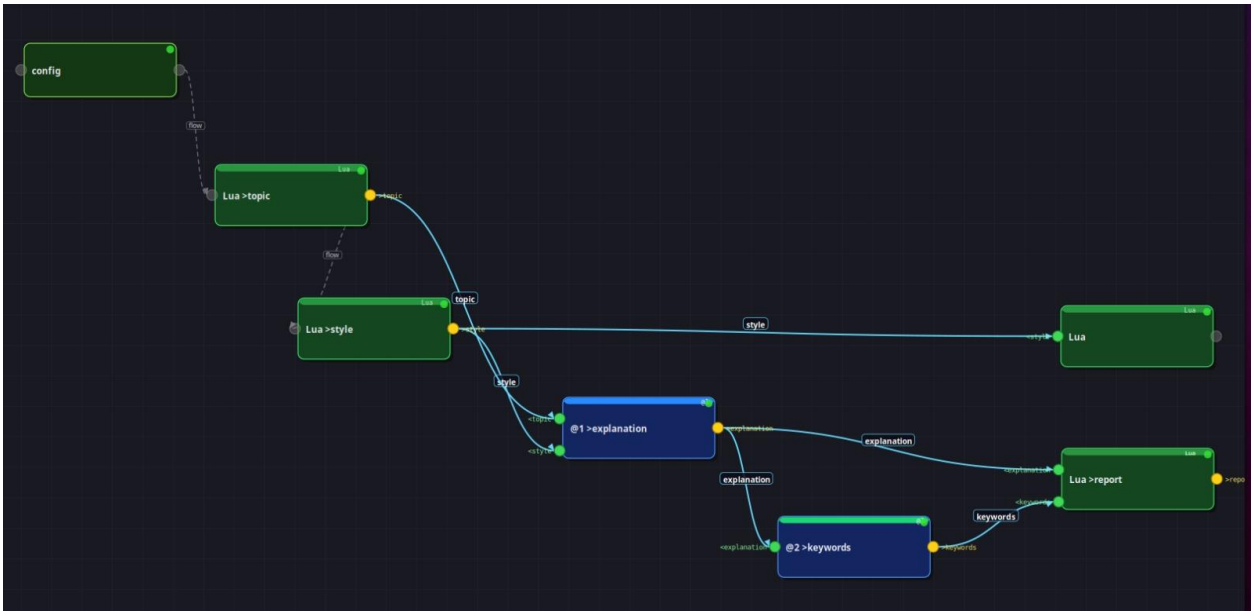
```
topic=black holes
style=poetic and accessible
max_words=50
```

```
io.write("black holes")
```

```
io.write("poetic and accessible")
```

```
io.write(io.open(os.getenv("MSH_VAR_style"),"r"):read("*all"))
```

```
local function read(v)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:clo
se()
    return t
end
local lines = {
    "=== Workflow Report ===",
    "Topic : " .. read("topic"),
    "Style : " .. read("style"),
    "",
    "Explanation:",
    read("explanation"),
    "",
    "Keywords: " .. read("keywords"),
}
print(table.concat(lines, "\n"))
```

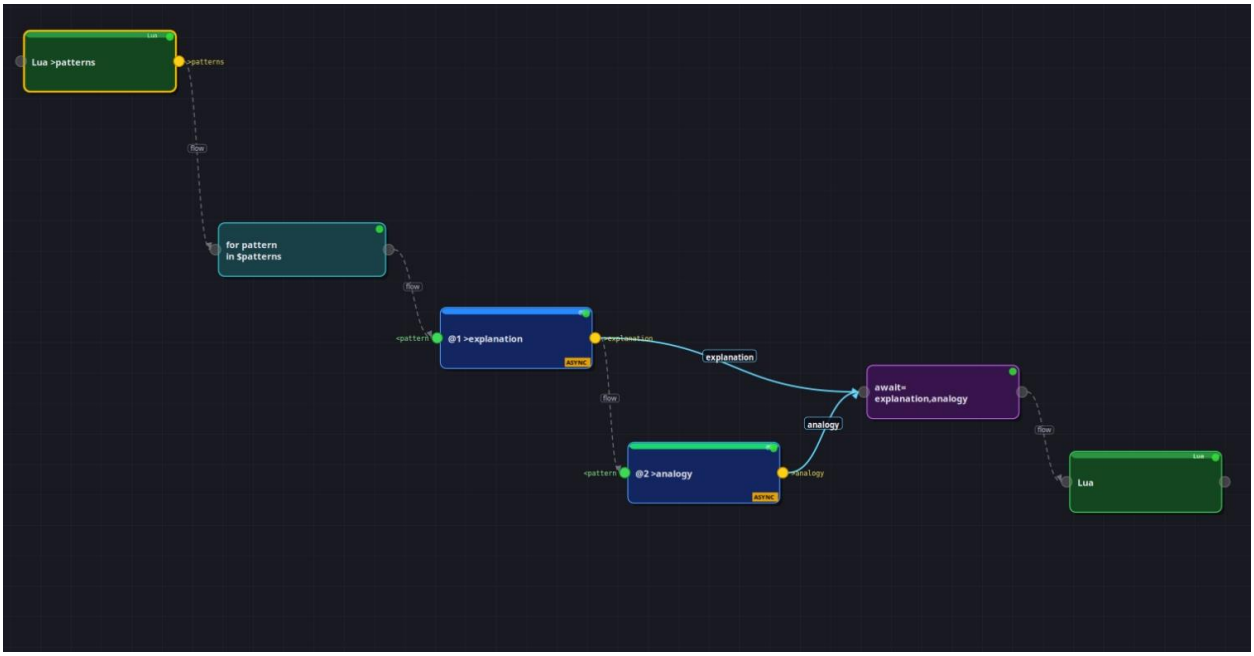


Pattern 18 — FOREACH + Async LLM: Parallel Lua Pattern Batch

What it does: Lua creates a list of Lua design patterns. FOREACH iterates. Per item two async LLMs generate explanation and analogy. Await + Lua prints each pair.

```
io.write("observer\ncoroutine\nprototype")
```

```
local function read(v)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:clo
se()
    return t
end
local pp = os.getenv("MSH_VAR_pattern")
local fp = io.open(pp,"r"); local name = fp:read("*l"):gsub("%s+$",""); fp:cl
ose()
print("=== " .. name .. " ===")
print("Explanation : " .. read("explanation"))
print("Analogy     : " .. read("analogy"))
print()

print("=== Batch complete ===")
```



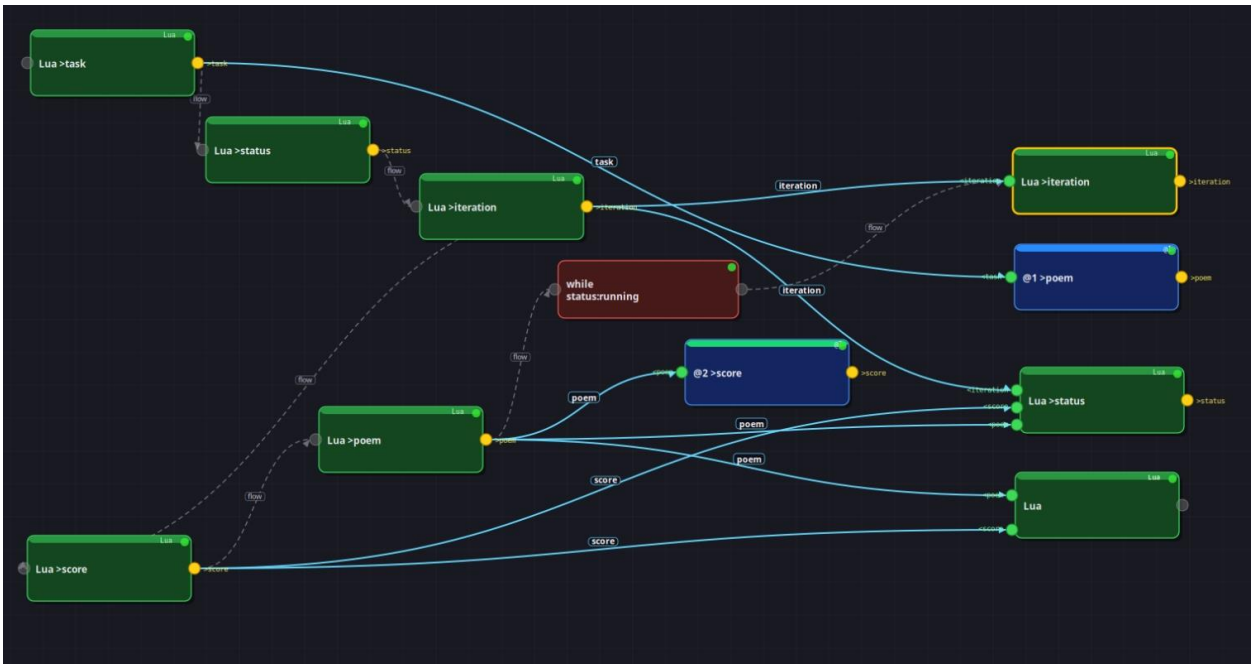
Pattern 19 — WHILE Quality Gate: Generate Lua Code Until Score >= 8

What it does: WHILE loop. Lua increments counter and writes status. LLM @1 generates Lua code. LLM @2 scores it 1–10. Lua checks threshold and writes done when score >= 8.

```
io.write("Write a haiku about the Lua programming language")

io.write("running")

io.write("0")

io.write("0")

io.write("")

local p = os.getenv("MSH_VAR_iteration")
local f = io.open(p,"r"); local v = tonumber(f:read("*l") or "0"); f:close()
local nv = v + 1
local fw = io.open(p,"w"); fw:write(tostring(nv)); fw:close()
print(nv)

local function read(v)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = (f:read("*l") or ""):gsub("%s+$","");
f:close()
    return t
end
local function write(v, val)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"w"); f:write(val); f:close()
end
local sc = read("score"):match("%d+") or "0"
```

```
print(string.format("[Iter %s] Score=%s", read("iteration"), sc))
print(read("poem")); print()
write("status", tonumber(sc) >= 8 and "done" or "running")
```

```
local function read(v)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:clo
se()
    return t
end
print("=== Accepted haiku (score=" .. read("score"):gsub("%s+","") .. ") ==="
)
print(read("poem"))
```



Pattern 20 — SPLIT + Async + MERGE: Map-Reduce Pipeline

What it does: Lua splits a text into three sentence chunks. Three async LLMs extract the main concept (MAP). Await barrier. LLM @2 synthesises a theme (REDUCE). Lua prints.

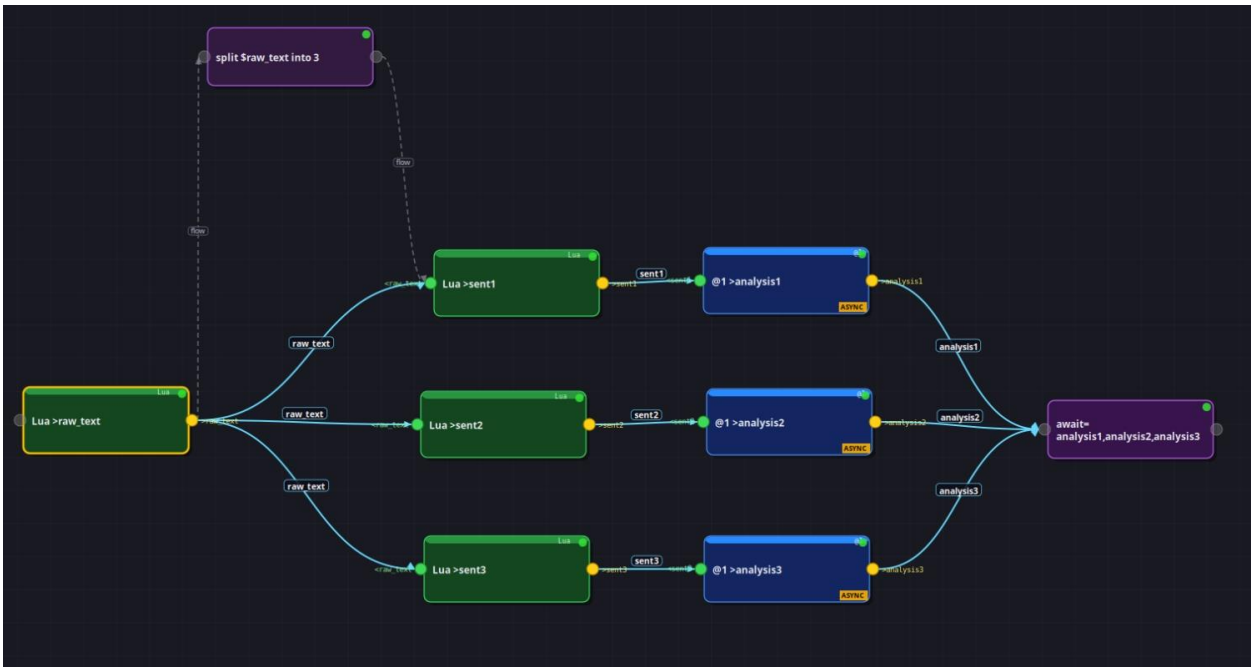
```
io.write("Lua was designed at PUC-Rio in Brazil in 1993. It is one of the fas
test scripting languages available. The language is embedded in millions of d
evices worldwide. Its coroutine system enables cooperative multitasking. Meta
tables give Lua a unique form of object-oriented programming.")
```

```
local p = os.getenv("MSH_VAR_raw_text")
local f = io.open(p,"r"); local text = f:read("*all"):gsub("%s+$",""); f:clos
e()
local sents = {}
for s in text:gmatch("[^%.]+%.") do table.insert(sents, s:match("^%s*(.-)%s*$
")) end
print(sents[1] or "")
```

```
local p = os.getenv("MSH_VAR_raw_text")
local f = io.open(p,"r"); local text = f:read("*all"):gsub("%s+$",""); f:clos
e()
local sents = {}
for s in text:gmatch("[^%.]+%.") do table.insert(sents, s:match("^%s*(.-)%s*$
")) end
local chunk = {}
for i = 2, 3 do if sents[i] then table.insert(chunk, sents[i]) end end
print(table.concat(chunk, ". "))
```

```
local p = os.getenv("MSH_VAR_raw_text")
local f = io.open(p,"r"); local text = f:read("*all"):gsub("%s+$",""); f:clos
e()
local sents = {}
for s in text:gmatch("[^%.]+%.") do table.insert(sents, s:match("^%s*(.-)%s*$
")) end
local chunk = {}
for i = 4, 5 do if sents[i] then table.insert(chunk, sents[i]) end end
print(table.concat(chunk, ". "))
```

```
local function read(v)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:clo
se()
    return t
end
print("=== Map ===")
print("Chunk 1: " .. read("analysis1"))
print("Chunk 2: " .. read("analysis2"))
print("Chunk 3: " .. read("analysis3"))
print(); print("=== Reduce ==="); print(read("summary"))
```



Pattern 21 — TRY/CATCH + LOOP: Resilient Lua Code Retry

What it does: LOOP max=3. LLM @1 generates Lua code. TRY executes it with `loadfile`. On failure CATCH records the error for the next self-correction iteration. Lua prints final status.

```
io.write("Write Lua code that parses '{\"name\": \"Alice\"}' with pattern mat
ching and prints the name field.")
```

```
io.write("fail")
```

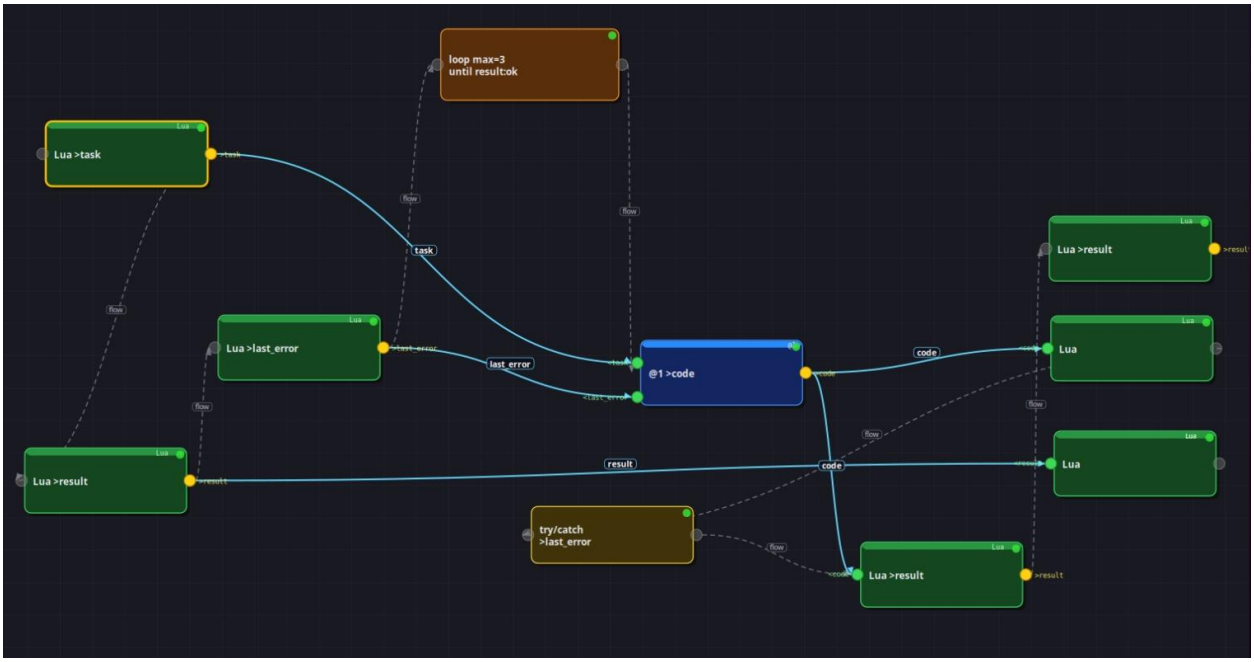
```
io.write("none")
```

```
local p = os.getenv("MSH_VAR_code")
local f = io.open(p,"r"); print("=== Generated Lua code ==="); print(f:read("
*all"):gsub("%s+$",""")); f:close()
```

```
local p = os.getenv("MSH_VAR_code")
local fn, load_err = loadfile(p)
if not fn then io.stderr:write(tostring(load_err).."\n"); os.exit(1) end
local ok_run, run_err = pcall(fn)
if not ok_run then io.stderr:write(tostring(run_err).."\n"); os.exit(1) end
print("ok")
```

```
print("=== Error on this attempt: try_block_failed ===")
io.write("fail")
```

```
local p = os.getenv("MSH_VAR_result")
local f = io.open(p,"r"); print("=== Final status: " .. f:read("*all"):gsub("
%s+$","") .. " ==="); f:close()
```



Pattern 22 — Multi-Variable Output: Structured Field Extraction

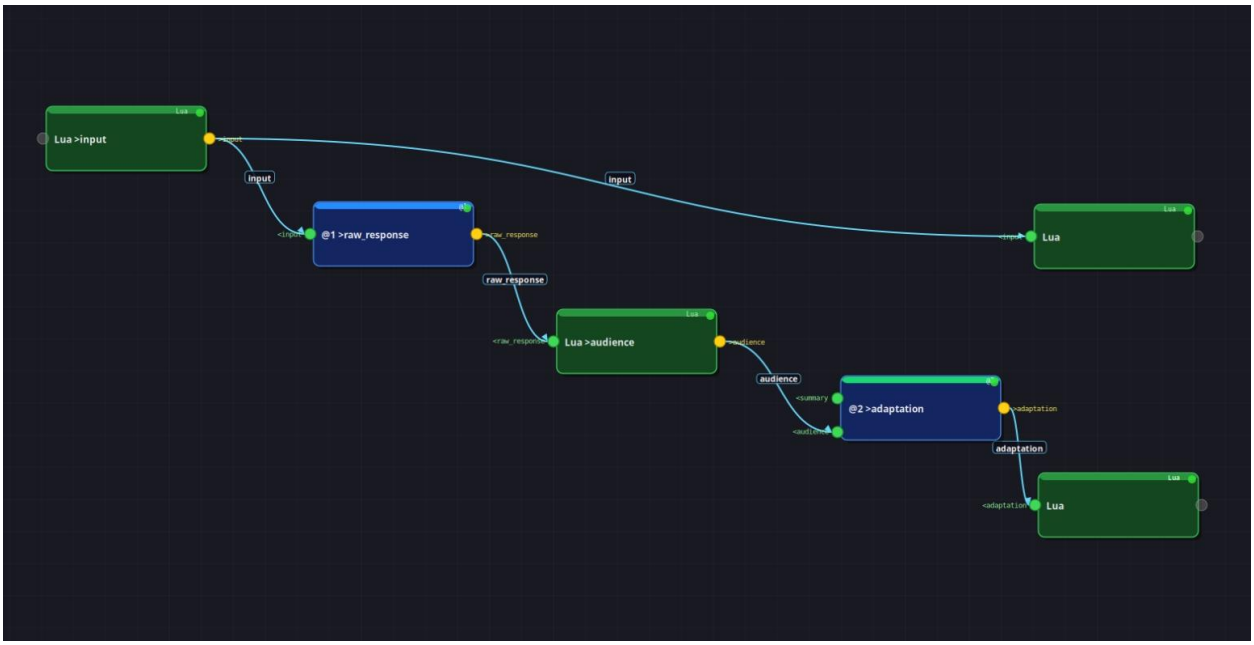
What it does: LLM @1 responds in strict 3-line format. Lua with 3 `>outvar` parses each line and writes fields directly to `MSH_VAR_*` files. LLM @2 adapts the summary for audience.

```
io.write("Coroutines in Lua allow cooperative multitasking by letting functio
ns pause and resume their execution, sharing a single thread without preempti
on.")
```

```
io.write(io.open(os.getenv("MSH_VAR_input"),"r"):read("*all"))
```

```
local p = os.getenv("MSH_VAR_raw_response")
local f = io.open(p,"r"); local text = f:read("*all"); f:close()
local function extract(label)
    return (text:match(label..":%s*(.-)%s*\n") or text:match(label..":%s*(.-)
%s*$") or "n/a")
end
local summary  = extract("SUMMARY")
local keywords = extract("KEYWORDS")
local audience = extract("AUDIENCE")
local function wvar(v, val)
    local wp = os.getenv("MSH_VAR_"..v)
    local wf = io.open(wp,"w"); wf:write(val); wf:close()
end
wvar("summary", summary); wvar("keywords", keywords); wvar("audience", audien
ce)
print("Summary  : " .. summary)
print("Keywords : " .. keywords)
print("Audience : " .. audience)
```

```
local p = os.getenv("MSH_VAR_adaptation")
local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
print(); print("=== Adapted version ==="); print(t)
```



Pattern 23 — CONFIG + WHILE + Multi-Model: Adaptive Pipeline

What it does: CONFIG documents parameters. WHILE loop runs until quality >= threshold. Lua increments counter. LLM @1 explains. LLM @2 scores. LLM @3 polishes final result.

```
subject=Lua coroutines
target_audience=high school student
quality_threshold=7
```

```
io.write("Lua coroutines")
```

```
io.write("high school student")
```

```
io.write("running")
```

```
io.write("0")
```

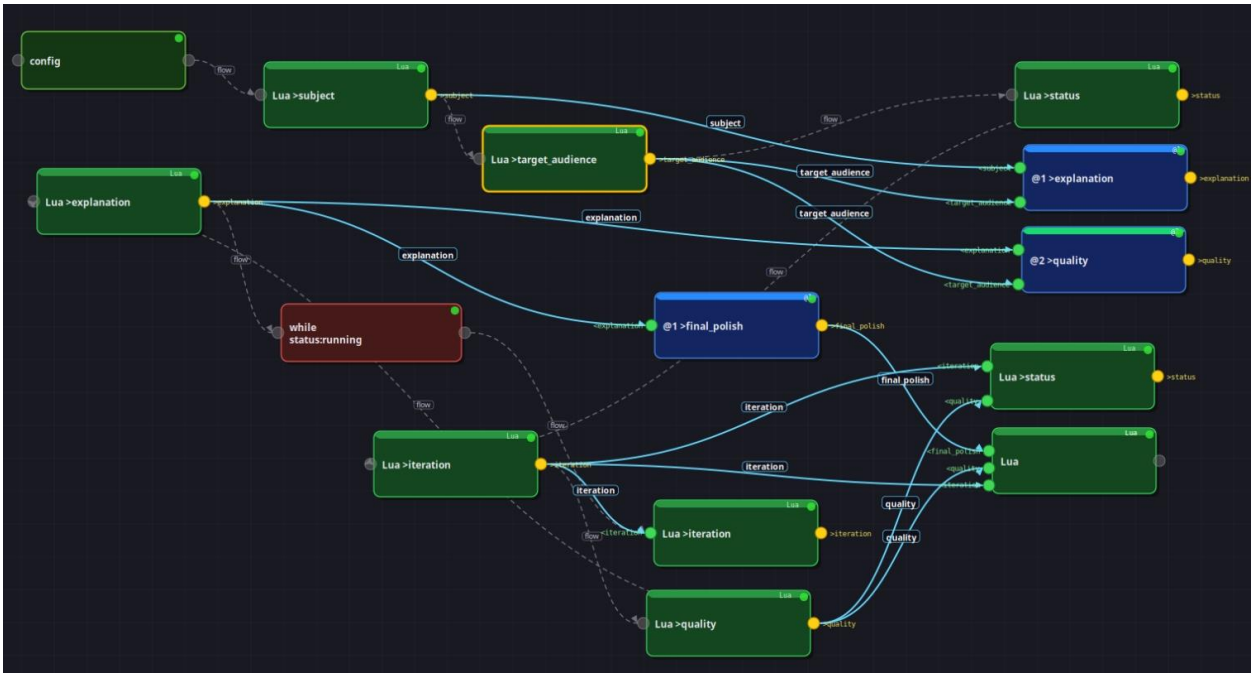
```
io.write("0")
```

```
io.write("")
```

```
local p = os.getenv("MSH_VAR_iteration")
local f = io.open(p,"r"); local v = tonumber(f:read("*l") or "0"); f:close()
local nv = v + 1
local fw = io.open(p,"w"); fw:write(tostring(nv)); fw:close()
print(nv)
```

```
local function read(v)
    local p = os.getenv("MSH_VAR_"..v)
```

```
    local f = io.open(p,"r"); local t = (f:read("*l") or ""):gsub("%s+$","");
f:close()
    return t
end
local function write(v, val)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"w"); f:write(val); f:close()
end
local q = read("quality"):match("%d+") or "0"
print(string.format("[Iter %s] Quality: %s", read("iteration"), q))
write("status", tonumber(q) >= 7 and "done" or "running")

local function read(v)
    local p = os.getenv("MSH_VAR_"..v)
    local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:clo
se()
    return t
end
print(string.format("=== Final (score=%s, iters=%s) ===",
    read("quality"):gsub("%s+",""), read("iteration"):gsub("%s+","")))
print(read("final_polish"))
```

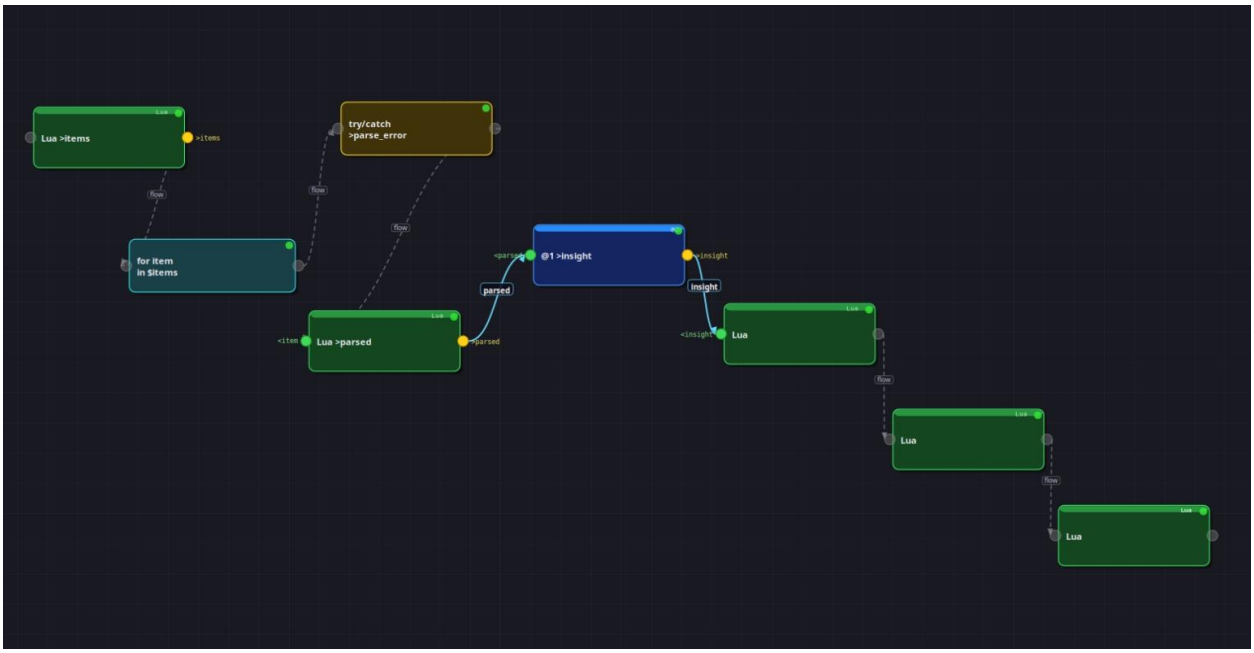


Pattern 24 — FOREACH + TRY/CATCH: Fault-Tolerant Batch Processing

What it does: Lua creates a list of key=value strings (one broken). FOREACH iterates. TRY block parses each with `string.match`. On parse failure CATCH prints `[ERR]`. Pipeline never stops.

```
io.write('x=1;y=nil\nx= broken !!\nx=nil;y=42')
```

```
local p = os.getenv("MSH_VAR_item")
local f = io.open(p,"r"); local raw = f:read("*l"):gsub("%s+$",""); f:close()
local fn, err = load("local " .. raw .. "; return {" .. raw .. "}")
if not fn then io.stderr:write("Parse error: "..tostring(err).."\n"); os.exit
(1) end
local ok, result = pcall(fn)
if not ok then io.stderr:write("Eval error: "..tostring(result).."\n"); os.ex
it(1) end
local parts = {}
for k, v in pairs(result) do
    if v ~= nil then table.insert(parts, tostring(k).."="..tostring(v)) end
end
print("Parsed OK: {" .. table.concat(parts, ", ") .. "}")

local p = os.getenv("MSH_VAR_insight")
local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
print("[OK] " .. t)

print("[ERR] Parse failed: try_block_failed")

print("=== Batch complete. Errors were isolated, pipeline never stopped. ==="
)
```



Quick Reference: Common Lua Patterns

Read single-line variable:

```
local f = io.open(os.getenv("MSH_VAR_name"), "r")
local val = f:read("*l"):gsub("%s+$", ""); f:close()
```

Write to multiple output variables:

```lua
local fw = io.open(os.getenv("MSH_VAR_field1"), "w")
fw:write(value1 .. "\n"); fw:close()
-- stdout goes to terminal only — not captured
```

WHILE exit:

```lua
local status = score >= threshold and "done" or "running"
local fs = io.open(os.getenv("MSH_VAR_status"), "w")
fs:write(status .. "\n"); fs:close()
```

LOOP exit (last stdout line):

```lua
print("ok")   -- must be absolute last line printed in loop body
```

Execute LLM-generated code safely:

```lua
local fn, err = loadfile(os.getenv("MSH_VAR_code"))
if fn then
    fn()
else
    io.stderr:write("Load error: " .. tostring(err) .. "\n")
    os.exit(1)
end
```

Appendix I: Code examples for each pattern

/home/igor > Sent to mshell (1215 bytes)

Received from GUI editor:

Pattern 1 Linear Data Pipeline (Lua)

```lua >raw

print("7")

```

```lua <raw >squared

local p = os.getenv("MSH_VAR_raw")

local f = io.open(p,"r"); local v = tonumber(f:read("*l")); f:close()

print(v * v)
```

```

```lua <squared >factorial
local p = os.getenv("MSH_VAR_squared")
local f = io.open(p,"r"); local n = tonumber(f:read("*l")); f:close()
local d = math.floor(n / 10) % 10
local fact = 1
for i = 2, d do fact = fact * i end
print(fact)
```

```lua <factorial >digit_sum
local p = os.getenv("MSH_VAR_factorial")
local f = io.open(p,"r"); local s = f:read("*l"):gsub("%s+",""); f:close()
local total = 0
for c in s:gmatch("%d") do total = total + tonumber(c) end
print(total)
```

```lua <digit_sum >reversed
local p = os.getenv("MSH_VAR_digit_sum")
local f = io.open(p,"r"); local s = f:read("*l"):gsub("%s+",""); f:close()
print(s:reverse())
```

```lua <reversed >report
local p = os.getenv("MSH_VAR_reversed")
local f = io.open(p,"r"); local v = f:read("*l"):gsub("%s+",""); f:close()
print("Pipeline complete. Final value: " .. v)
```
```

```lua <report
local p = os.getenv("MSH_VAR_report")
local f = io.open(p,"r")
print("=== Pipeline Result ===")
print(f:read("*all"):gsub("%s+$",""))
f:close()
```

7

49

24

6

6

Pipeline complete. Final value: 6

=== Pipeline Result ===

Pipeline complete. Final value: 6 1

/home/igor > Sent to mshell (798 bytes)

Received from GUI editor:

Pattern 2 LLM in the Middle

```lua >data
print("City temperatures (°C): Oslo=-4, Reykjavik=-1, London=9, Rome=18, Cairo=28, Bangkok=34")
```

```lua <data
io.write(io.open(os.getenv("MSH_VAR_data"),"r"):read("*all"))
```

<!--@1 <data >analysis

Analyze this temperature data briefly. Identify the hottest and coldest city,

describe the spread, and suggest a travel tip. Two sentences max.

-->

```lua <analysis

local p = os.getenv("MSH_VAR_analysis")

local f = io.open(p,"r"); local text = f:read("*all"):gsub("%s+$",""); f:close()

local words = 0

for _ in text:gmatch("%S+") do words = words + 1 end

local sents = 0

for _ in text:gmatch("[%.!%?]") do sents = sents + 1 end

print(string.format("LLM response: %d words, ~%d sentence(s)", words, sents))

print("--- Analysis ---")

print(text)

---

City temperatures (°C): Oslo=-4, Reykjavik=-1, London=9, Rome=18, Cairo=28, Bangkok=34

City temperatures (°C): Oslo=-4, Reykjavik=-1, London=9, Rome=18, Cairo=28, Bangkok=34

Bangkok is the hottest at 34°C while Oslo is the coldest at -4°C, creating a 38-degree temperature spread that clearly shows the progression from northern European winter conditions to tropical Southeast Asian heat. For travelers planning multi-city trips across this range, pack layers and consider starting in the colder cities and moving south to avoid the shock of dramatic temperature changes.

---

/home/igor > Sent to mshell (849 bytes)

Received from GUI editor:

---

# Pattern 3  Fan-Out: One Variable - Many Consumers
```

```lua >sentence
print("The wandering moon crossed the silent silver river at midnight")
```

```lua <sentence
local p = os.getenv("MSH_VAR_sentence")
local f = io.open(p,"r"); local s = f:read("*l"):gsub("%s+$",""); f:close()
local vowels = 0
for _ in s:lower():gmatch("[aeiou]") do vowels = vowels + 1 end
print("Lua consumer 1 \u2014 vowel count: " .. vowels)
```

```lua <sentence
local p = os.getenv("MSH_VAR_sentence")
local f = io.open(p,"r"); local s = f:read("*l"):gsub("%s+$",""); f:close()
local words = {}
for w in s:gmatch("%S+") do table.insert(words, 1, w) end
print("Lua consumer 2 \u2014 reversed words: " .. table.concat(words, " "))
```

<!--@1 <sentence
Identify the mood of this sentence in one word, then count its syllables approximately.
Reply in exactly one sentence.
-->

The mood is mystical, and the sentence contains approximately 17 syllables.

/home/igor >

Sent to mshell (663 bytes)

Received from GUI editor:

Pattern 4 LLM Code Generation - Execute via Variable

```lua >task
print("Write a Lua function called primes(n) that returns the first n primes, call it with n=8, print each result.")
```

<!--@1 <task >code
Write only Lua code. No markdown fences, no explanation.
The code must be self-contained and runnable with lua5.4.
-->

```lua <code
local code_path = os.getenv("MSH_VAR_code")
local cf = io.open(code_path,"r"); local src = cf:read("*all"); cf:close()
print("=== Generated Lua code ==="); print(src:gsub("%s+$",""))
print("=== Executing ===")
local fn, err = loadfile(code_path)
if fn then fn() else print("Load error: " .. tostring(err)) end
```

Write a Lua function called primes(n) that returns the first n primes, call it with n=8, print each result.

```
function primes(n)
  if n <= 0 then
    return {}
  end
  local result = {}
  local num = 2
  while #result < n do
```

```
    local is_prime = true
    for i = 2, math.sqrt(num) do
      if num % i == 0 then
        is_prime = false
        break
      end
    end
    if is_prime then
      table.insert(result, num)
    end
    num = num + 1
  end
  return result
end
local first_eight_primes = primes(8)
for i, prime in ipairs(first_eight_primes) do
  print(prime)
end
=== Generated Lua code ===
function primes(n)
  if n <= 0 then
    return {}
  end
  local result = {}
  local num = 2
  while #result < n do
    local is_prime = true
```

```
    for i = 2, math.sqrt(num) do
      if num % i == 0 then
        is_prime = false
        break
      end
    end
    if is_prime then
      table.insert(result, num)
    end
    num = num + 1
  end
  return result
end
local first_eight_primes = primes(8)
for i, prime in ipairs(first_eight_primes) do
  print(prime)
end    0
=== Executing ===
2
3
5
7
11
13
17
19
---
```

/home/igor > Sent to mshell (1261 bytes)

Received from GUI editor:

Pattern 5 Two-LLM Review Chain

```lua >task
print("write a Lua function that computes the nth Fibonacci number using memoization")
```

```lua <task
io.write(io.open(os.getenv("MSH_VAR_task"),"r"):read("*all"))
```

<!--@1 <task >code

Write only Lua code, no fences, no explanation.

The code must define the function, call it for n=10, and print the result.

-->

```lua <code
local p = os.getenv("MSH_VAR_code")
local f = io.open(p,"r"); print("=== Model 1 generated ===");
print(f:read("*all"):gsub("%s+$",""")); f:close()
```

<!--@2 <code >review

Review this Lua code. Is the memoization correct and efficient? Reply in 2 sentences max.

-->

```lua <review
local p = os.getenv("MSH_VAR_review")
local f = io.open(p,"r"); print("=== Model 2 review ===");
print(f:read("*all"):gsub("%s+$",""")); f:close()
```

<!--@1 <code <review >improved

Improve the Lua code based on the review.

Return ONLY the improved Lua code, no fences, no explanation.

-->

```lua <improved
local p = os.getenv("MSH_VAR_improved")
local f = io.open(p,"r"); local src = f:read("*all"); f:close()
print("=== Final improved code ==="); print(src:gsub("%s+$",""))
print("=== Executing ===")
local fn, err = loadfile(p)
if fn then fn() else print("Error: " .. tostring(err)) end
```

---

write a Lua function that computes the nth Fibonacci number using memoization

write a Lua function that computes the nth Fibonacci number using memoization

```lua
local memo = {}
function fibonacci(n)
  if n <= 1 then
    return n
  end
  if memo[n] then
    return memo[n]
  end
  memo[n] = fibonacci(n - 1) + fibonacci(n - 2)
  return memo[n]
end

print(fibonacci(10))
```

=== Model 1 generated ===

```
local memo = {}

function fibonacci(n)
   if n <= 1 then
      return n
   end
   if memo[n] then
      return memo[n]
   end
   memo[n] = fibonacci(n - 1) + fibonacci(n - 2)
   return memo[n]
end

print(fibonacci(10))  0
```

Yes, the memoization is correct: it caches results in `memo[n]` and avoids recomputing overlapping subproblems. For computing a single `fibonacci(n)`, this turns the time complexity from exponential to linear in `n`, and the overhead of the table lookups is minimal, so it\u2019s efficient for typical `n`.

=== Model 2 review ===

Yes, the memoization is correct: it caches results in `memo[n]` and avoids recomputing overlapping subproblems. For computing a single `fibonacci(n)`, this turns the time complexity from exponential to linear in `n`, and the overhead of the table lookups is minimal, so it\u2019s efficient for typical `n`. 0

```
local memo = {[0] = 0, [1] = 1}

function fibonacci(n)
   if memo[n] then
      return memo[n]
   end
   memo[n] = fibonacci(n - 1) + fibonacci(n - 2)
   return memo[n]
```

```
end
print(fibonacci(10))
=== Final improved code ===
local memo = {[0] = 0, [1] = 1}
function fibonacci(n)
  if memo[n] then
    return memo[n]
  end
  memo[n] = fibonacci(n - 1) + fibonacci(n - 2)
  return memo[n]
end
print(fibonacci(10))  0
=== Executing ===
55
/home/igor >
```

```
/home/igor > Sent to mshell (719 bytes)
Received from GUI editor:
```

```
# Pattern 6 Parallel 3-Model Query
```lua >question
print("If you could remove one cognitive bias from humanity, which would it be and why? One sentence.")
```
```lua <question
io.write(io.open(os.getenv("MSH_VAR_question"),"r"):read("*all"))
```
```

<!--@1 <question >ans1

Answer in exactly one sentence.

-->

<!--@2 <question >ans2

Answer in exactly one sentence.

-->

<!--@3 <question >ans3

Answer in exactly one sentence.

-->

```lua <ans1 <ans2 <ans3

for i, v in ipairs({"ans1","ans2","ans3"}) do

  local p = os.getenv("MSH_VAR_"..v)

  local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()

  print(string.format("=== Model %d (%d chars) ===", i, #t))

  print(t); print()

end

```

If you could remove one cognitive bias from humanity, which would it be and why? One sentence.

If you could remove one cognitive bias from humanity, which would it be and why? One sentence.

I would remove confirmation bias because it prevents people from updating their beliefs when presented with contradictory evidence, which undermines scientific progress, rational decision-making, and constructive dialogue across differences.

I would remove confirmation bias, because it distorts how we gather and interpret information, fuels polarization, and undermines our ability to learn, collaborate, and correct mistakes.

I would remove the fundamental attribution error because it fuels prejudice and conflict by making people blame others' behavior on character while excusing their own with circumstances.

=== Model 1 (241 chars) ===

I would remove confirmation bias because it prevents people from updating their beliefs when presented with contradictory evidence, which undermines scientific progress, rational decision-making, and constructive dialogue across differences.

=== Model 2 (186 chars) ===

I would remove confirmation bias, because it distorts how we gather and interpret information, fuels polarization, and undermines our ability to learn, collaborate, and correct mistakes.

=== Model 3 (186 chars) ===

I would remove the fundamental attribution error because it fuels prejudice and conflict by making people blame others' behavior on character while excusing their own with circumstances.

/home/igor > Sent to mshell (1201 bytes)

Received from GUI editor:

Pattern 7 Evaluator-Optimizer Loop

```lua >task

print("write a Lua function is_palindrome(s) ignoring spaces and case, test on 3 examples")

```

<!--@loop max=3 until=verdict:ACCEPTED-->

<!--@1 <task >code

Write only Lua code, no fences, no explanation. Self-contained with test output.

-->

```lua <code

local p = os.getenv("MSH_VAR_code")
```

local f = io.open(p,"r"); print("=== Generated code ===");
print(f:read("*all"):gsub("%s+$",""")); f:close()

```

<!--@2 <code >verdict

Review this Lua code for correctness and idiomatic style.

If ALL checks pass, reply with ONLY one word: ACCEPTED

If any check fails, reply with ONLY one word: REJECTED

Do NOT add any explanation, punctuation, or extra text \u2014 just the single word.

-->

```lua <verdict

local p = os.getenv("MSH_VAR_verdict")

local f = io.open(p,"r"); print("=== Verdict ==="); print(f:read("*all"):gsub("%s+$",""));
f:close()

```

<!--@end_loop-->

```lua <code

local p = os.getenv("MSH_VAR_code")

print("=== Final accepted code ===")

local f = io.open(p,"r"); print(f:read("*all"):gsub("%s+$",""")); f:close()

print("=== Executing ===")

local fn, err = loadfile(p)

if fn then fn() else print("Load error: " .. tostring(err)) end

```

---

write a Lua function is_palindrome(s) ignoring spaces and case, test on 3 examples

[loop] Starting loop: max=3 until=verdict:ACCEPTED

function is_palindrome(s)

  local cleaned = string.gsub(string.lower(s), "%s", "")
```

```
  local reversed = string.reverse(cleaned)
  return cleaned == reversed
end
print("racecar:", is_palindrome("racecar"))
print("A man a plan a canal Panama:", is_palindrome("A man a plan a canal Panama"))
print("hello world:", is_palindrome("hello world"))
=== Generated code ===
function is_palindrome(s)
  local cleaned = string.gsub(string.lower(s), "%s", "")
  local reversed = string.reverse(cleaned)
  return cleaned == reversed
end
print("racecar:", is_palindrome("racecar"))
print("A man a plan a canal Panama:", is_palindrome("A man a plan a canal Panama"))
print("hello world:", is_palindrome("hello world"))        0
REJECTED
=== Verdict ===
REJECTED      0
[loop] until check: last_line='REJECTED' expected='ACCEPTED'
[loop] Iteration 1/3 \u2014 condition not met, looping back
function is_palindrome(s)
  local cleaned = string.gsub(string.lower(s), "%s", "")
  local reversed = string.reverse(cleaned)
  return cleaned == reversed
end
print("racecar:", is_palindrome("racecar"))
print("A man a plan a canal Panama:", is_palindrome("A man a plan a canal Panama"))
```

```
print("hello world:", is_palindrome("hello world"))
=== Generated code ===
function is_palindrome(s)
    local cleaned = string.gsub(string.lower(s), "%s", "")
    local reversed = string.reverse(cleaned)
    return cleaned == reversed
end
print("racecar:", is_palindrome("racecar"))
print("A man a plan a canal Panama:", is_palindrome("A man a plan a canal Panama"))
print("hello world:", is_palindrome("hello world"))          0
ACCEPTED
=== Verdict ===
ACCEPTED      0
[loop] Exiting loop after 2 iteration(s). reason: until condition met
=== Final accepted code ===
function is_palindrome(s)
    local cleaned = string.gsub(string.lower(s), "%s", "")
    local reversed = string.reverse(cleaned)
    return cleaned == reversed
end
print("racecar:", is_palindrome("racecar"))
print("A man a plan a canal Panama:", is_palindrome("A man a plan a canal Panama"))
print("hello world:", is_palindrome("hello world"))          0
=== Executing ===
racecar:          true
A man a plan a canal Panama:          true
hello world:    false
```

/home/igor >

/home/igor > Sent to mshell (1187 bytes)

Received from GUI editor:

Pattern 8 Multi-Stage Lua + Multi-Model Pipeline

```lua >raw_data
math.randomseed(42)
local nums = {}
for i = 1, 10 do table.insert(nums, math.random(0,99)) end
print(table.concat(nums, ","))
```

```lua <raw_data >stats
local p = os.getenv("MSH_VAR_raw_data")
local f = io.open(p,"r"); local line = f:read("*l"):gsub("%s+$",""); f:close()
local nums = {}
for n in line:gmatch("%d+") do table.insert(nums, tonumber(n)) end
local sum = 0; local mn, mx = nums[1], nums[1]
for _, v in ipairs(nums) do
  sum = sum + v
  if v < mn then mn = v end
  if v > mx then mx = v end
end
print(string.format("count=%d mean=%.1f min=%d max=%d sum=%d", #nums,
sum/#nums, mn, mx, sum))
```

<!--@1 <stats >analysis

In one sentence, describe what you can infer about the spread and central tendency.

-->

<!--@2 <analysis >headline

Compress this analysis into a 5-word tweet-style headline. Nothing else.

-->

```lua <raw_data <stats <analysis <headline
for _, v in ipairs({"raw_data","stats","analysis","headline"}) do
  local p = os.getenv("MSH_VAR_"..v)
  local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
  print(string.format("[%s]: %s", v, t))
end
```

49,75,85,53,63,6,24,2,23,23

count=10 mean=40.3 min=2 max=85 sum=403

The data shows a moderate central tendency around 40.3 with high variability, as evidenced by the large range of 83 (from 2 to 85) relative to the mean, suggesting the distribution is likely skewed or contains outliers.

High variability around 40.3 mean

[raw_data]: 49,75,85,53,63,6,24,2,23,23

[stats]: count=10 mean=40.3 min=2 max=85 sum=403

[analysis]: The data shows a moderate central tendency around 40.3 with high variability, as evidenced by the large range of 83 (from 2 to 85) relative to the mean, suggesting the distribution is likely skewed or contains outliers.

[headline]: High variability around 40.3 mean

/home/igor > Sent to mshell (703 bytes)

Received from GUI editor:

Pattern 9 Routing: LLM Classifies - Conditional Lua Branch

```lua >input1
print("calculate the sum of squares from 1 to 10")
```

```lua <input1
io.write(io.open(os.getenv("MSH_VAR_input1"),"r"):read("*all"))
```

<!--@1 <input1 >route1
Classify this task into exactly one word: MATH, CODE, or TEXT.
-->

```lua <route1 if=route1:MATH
print("=== MATH branch ===")
local total = 0
for i = 1, 10 do total = total + i*i end
print("Sum of squares 1..10 = " .. total)
```

<!--@1 <input1 if=route1:TEXT
Answer this in plain language, no code.
-->

```lua <route1
local p = os.getenv("MSH_VAR_route1")
local f = io.open(p,"r"); print("Classified as: " .. f:read("*l"):gsub("%s+$","")); f:close()
```

---

calculate the sum of squares from 1 to 10

calculate the sum of squares from 1 to 10

MATH

=== MATH branch ===

Sum of squares 1..10 = 385

Classified as: MATH

/home/igor > Sent to mshell (1438 bytes)

Received from GUI editor:

Pattern 10 Full Pipeline: All Patterns Combined

```lua >raw_data
local t = {}
for i = 1, 10 do table.insert(t, i*i) end
print(table.concat(t, " "))
```

```lua <raw_data >stats
local p = os.getenv("MSH_VAR_raw_data")
local f = io.open(p,"r"); local line = f:read("*l"):gsub("%s+$",""); f:close()
local nums = {}
for n in line:gmatch("%d+") do table.insert(nums, tonumber(n)) end
local sum = 0
for _, v in ipairs(nums) do sum = sum + v end
print(string.format("First 10 perfect squares: %s", line))
print(string.format("Sum=%d  Mean=%.1f", sum, sum/#nums))
```

<!--@1 <stats >analysis

In one sentence, describe what is mathematically interesting about the first 10 perfect squares.

-->

<!--@2 <raw_data >poem

Write a 2-line poem about perfect squares.

-->

```lua <analysis <poem
local function read(v)
  local p = os.getenv("MSH_VAR_"..v)
  local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
  return t
end
print("=== Analysis ==="); print(read("analysis"))
print(); print("=== Poem ==="); print(read("poem"))
```

<!--@1 <analysis <poem >combined

Combine the mathematical analysis and the poem into one elegant sentence.

-->

```lua <combined
local p = os.getenv("MSH_VAR_combined")
local f = io.open(p,"r"); local text = f:read("*all"):gsub("%s+$",""); f:close()
local border = string.rep("\u2550", math.min(#text, 72))
print("=== Final Output ==="); print(border); print(text); print(border)
```

---

1 4 9 16 25 36 49 64 81 100

First 10 perfect squares: 1 4 9 16 25 36 49 64 81 100

Sum=385 Mean=38.5

The first 10 perfect squares demonstrate rapid growth with increasing gaps between consecutive terms (the differences form the sequence 3, 5, 7, 9, 11, 13, 15, 17, 19),

illustrating how quadratic functions accelerate and creating a sum of 385 that equals n(n+1)(2n+1)/6 where n=10.

Numbers step by step in ordered pairs,

Quiet miracles of counting, blooming into perfect squares.

=== Analysis ===

The first 10 perfect squares demonstrate rapid growth with increasing gaps between consecutive terms (the differences form the sequence 3, 5, 7, 9, 11, 13, 15, 17, 19), illustrating how quadratic functions accelerate and creating a sum of 385 that equals n(n+1)(2n+1)/6 where n=10.

=== Poem ===

Numbers step by step in ordered pairs,

Quiet miracles of counting, blooming into perfect squares.

The perfect squares reveal their quiet miracle through accelerating gaps of consecutive odd numbers (3, 5, 7, ..., 19), blooming into the elegant formula n(n+1)(2n+1)/6 that captures their cumulative beauty as 385.

=== Final Output ===

\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550

The perfect squares reveal their quiet miracle through accelerating gaps of consecutive odd numbers (3, 5, 7, ..., 19), blooming into the elegant formula n(n+1)(2n+1)/6 that captures their cumulative beauty as 385.

\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550

/home/igor > Sent to mshell (698 bytes)

Received from GUI editor:

Pattern 11 MShell Node with Multiple Models

```lua >topic
print("quantum computing")
```

```lua >style
print("vivid and analogy-driven")
```

```mshell <topic <style >explanation
ollama1 "Explain $topic in a $style way in one sentence"
```

```mshell <explanation >keywords
ollama2 "Extract exactly 3 keywords from: $explanation. Reply with 3 words, comma-separated only."
```

```lua <explanation <keywords
local function read(v)
  local p = os.getenv("MSH_VAR_"..v)
  local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
  return t
end
print("=== Explanation ==="); print(read("explanation"))
print(); print("=== Keywords ==="); print(read("keywords"))
```

---

quantum computing

vivid and analogy-driven

Quantum computing is like having a magical coin that can be heads, tails, and spinning simultaneously, allowing you to explore every path through a maze at once instead of trudging through each route one by one.Quantum,computing,maze=== Explanation ===

Quantum computing is like having a magical coin that can be heads, tails, and spinning simultaneously, allowing you to explore every path through a maze at once instead of trudging through each route one by one.

=== Keywords ===

Quantum,computing,maze

/home/igor > Sent to mshell (839 bytes)

Received from GUI editor:

Pattern 12 Async Parallel 3 Models + Await Barrier + Synthesis

```lua >question

print("What are metatables and metamethods in Lua?")

```

```lua <question

io.write(io.open(os.getenv("MSH_VAR_question"),"r"):read("*all"))

```

<!--@1 <question >ans1 async

Explain in one sentence for a complete beginner.

-->

<!--@2 <question >ans2 async

Explain in one sentence using a real-world analogy.

-->

<!--@3 <question >ans3 async

Explain in one sentence using precise technical terminology.

-->

```lua await=ans1,ans2,ans3
```

<!--@1 <ans1 <ans2 <ans3 >final

Synthesize these three explanations into one perfect sentence for all audiences.

-->

```lua <final
local p = os.getenv("MSH_VAR_final")
local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
print("=== Three perspectives synthesized ==="); print(t)
```

What are metatables and metamethods in Lua?

What are metatables and metamethods in Lua?

[async llm] Launched PID 59907 \u2192 var=ans1 (model @1)

[async llm] Launched PID 59908 \u2192 var=ans2 (model @2)

[async llm] Launched PID 59910 \u2192 var=ans3 (model @3)

[async] await= barrier: waiting for vars: ans1,ans2,ans3

[async] Waiting for PID 59907 (var=ans1)...

[async] PID 59907 done (var=ans1)

Metatables in Lua are special tables that let you customize how regular tables behave when you perform operations like addition, subtraction, or indexing on them, using special functions called metamethods.

Just as Lua's metatables allow us to redefine how data behaves through custom rules, removing cognitive biases like confirmation bias or fundamental attribution error would

let us redefine how our minds process information, enabling more accurate, fair, and collaborative human behavior.

=== Three perspectives synthesized ===

Just as Lua's metatables allow us to redefine how data behaves through custom rules, removing cognitive biases like confirmation bias or fundamental attribution error would let us redefine how our minds process information, enabling more accurate, fair, and collaborative human behavior.

/home/igor > Sent to mshell (1347 bytes)

Received from GUI editor:

Pattern 13 WHILE Loop: Iterative Counter with LLM Commentary

```lua >status
print("running")
```

```lua >counter
print("0")
```

<!--@while status:running-->

```lua <counter <status >counter >status
local function read(v)
  local p = os.getenv("MSH_VAR_"..v)
  local f = io.open(p,"r"); local t = f:read("*all") or "0"; f:close()
  return t:gsub("%s+","")
end
local function write(v, val)
  local p = os.getenv("MSH_VAR_"..v)
  local f = io.open(p,"w"); f:write(val); f:close()
end
```

local raw = read("counter")

local val = tonumber(raw:match("%d+") or "0") + 1

write("counter", tostring(val))

write("status", val >= 3 and "done" or "running")

print(val)

```

<!--@1 <counter >comment

The input contains a number. In one sentence, say something mathematically interesting about it.

-->

```lua <comment

local pc = os.getenv("MSH_VAR_counter")

local fc = io.open(pc,"r"); local cnt = (fc:read("*all") or ""):gsub("%s+",""); fc:close()

local pm = os.getenv("MSH_VAR_comment")

local fm = io.open(pm,"r"); local msg = fm:read("*all"):gsub("%s+$",""); fm:close()

print(string.format("[iter %s] %s", cnt, msg))

```

<!--@end_while-->

```lua <counter

local p = os.getenv("MSH_VAR_counter")

local f = io.open(p,"r"); local v = (f:read("*all") or ""):gsub("%s+",""); f:close()

print("=== WHILE done. Final counter = " .. v .. " ===")

```

---

running

0

[while] iteration 1 \u2014 condition met, executing body
```

1

The number 1 is the multiplicative identity, meaning any number multiplied by 1 equals itself, and it's the only positive integer that is neither prime nor composite.

[iter 1] The number 1 is the multiplicative identity, meaning any number multiplied by 1 equals itself, and it's the only positive integer that is neither prime nor composite.

[while] iteration 2 \u2014 condition met, executing body

2

The number 2 is the only even prime number, serving as the fundamental building block of binary systems and the base of exponential growth in computer science.

[iter 2] The number 2 is the only even prime number, serving as the fundamental building block of binary systems and the base of exponential growth in computer science.

[while] iteration 3 \u2014 condition met, executing body

3

The number 3 is the smallest odd prime and the only number that equals the sum of all smaller positive integers (1 + 2 = 3), making it fundamental to triangular numbers and geometric tessellations.

[iter 3] The number 3 is the smallest odd prime and the only number that equals the sum of all smaller positive integers (1 + 2 = 3), making it fundamental to triangular numbers and geometric tessellations.

[while] condition 'status:running' not met, exiting after 3 iter

=== WHILE done. Final counter = 3 ===

/home/igor > Sent to mshell (697 bytes)

Received from GUI editor:

Pattern 14 FOREACH: LLM Explains Each Lua Library

```lua >libraries

io.write("io\nmath\nstring")

```

<!--@foreach lib in libraries-->

<!--@1 <lib >description

The input is a Lua standard library name. In one sentence, describe its best use case

and give one example function name.

-->

```lua <description
local pl = os.getenv("MSH_VAR_lib")
local fl = io.open(pl,"r"); local lib = fl:read("*l"):gsub("%s+$",""); fl:close()
local pd = os.getenv("MSH_VAR_description")
local fd = io.open(pd,"r"); local desc = fd:read("*all"):gsub("%s+$",""); fd:close()
print("--- " .. lib .. " ---"); print(desc); print()
```

<!--@end_foreach-->

```lua
print("=== All Lua libraries described ===")
```

---

io

math

string[foreach] iter 1: lib=io

The io library is best used for file operations and input/output tasks, with io.open() being essential for reading from and writing to files.

--- io ---

The io library is best used for file operations and input/output tasks, with io.open() being essential for reading from and writing to files.

[foreach] iter 2: lib=math

The math library is best used for mathematical computations and scientific calculations, with math.sqrt() being essential for computing square roots and geometric operations.

--- math ---

The math library is best used for mathematical computations and scientific calculations, with math.sqrt() being essential for computing square roots and geometric operations.

[foreach] iter 3: lib=string

The string library is best used for text processing and manipulation tasks, with string.match() being essential for pattern matching and data extraction from strings.

--- string ---

The string library is best used for text processing and manipulation tasks, with string.match() being essential for pattern matching and data extraction from strings.

=== All Lua libraries described ===

/home/igor > Sent to mshell (934 bytes)

Received from GUI editor:

Pattern 15 TRY/CATCH: Safe Execution with Error Capture

```lua >input
print("hello Lua world")
```

<!--@try-->

```lua <input >result
local p = os.getenv("MSH_VAR_input")
local f = io.open(p,"r"); local text = f:read("*l"):gsub("%s+$",""); f:close()
local broken = nil
local _ = broken.field   -- intentional nil index error
print(text)
os.exit(1)
```

<!--@catch >error-->

```lua
print("=== Caught error: try_block_failed ===")
print("Pipeline continues safely.")
```

<!--@end_try-->

```lua <input >safe_result
local p = os.getenv("MSH_VAR_input")
local f = io.open(p,"r"); local text = f:read("*l"):gsub("%s+$",""); f:close()
local count = 0
for _ in text:gmatch("%S+") do count = count + 1 end
print("Safe fallback: word count = " .. count)
```

```lua <safe_result
local p = os.getenv("MSH_VAR_safe_result")
local f = io.open(p,"r"); print("=== Safe result ==="); print((f:read("*all"):gsub("%s+$","")));
f:close()
```

---

hello Lua world

[try] executing try block

lua5.4: /tmp/mshell_script_47725.lua:4: attempt to index a nil value (local 'broken')

stack traceback:

	/tmp/mshell_script_47725.lua:4: in main chunk

	[C]: in ?

[try] try block failed, executing catch block

=== Caught error: try_block_failed ===

Pipeline continues safely.

Safe fallback: word count = 3

=== Safe result ===

Safe fallback: word count = 3

/home/igor > Sent to mshell (1110 bytes)

Received from GUI editor:

Pattern 16 SPLIT + MERGE: Divide-and-Conquer Analysis

```lua >dataset
io.write("AAPL:182,179,185,188,191\nMSFT:374,371,378,382,385")
```

<!--@split dataset into 2-->

<!--@1 <dataset_1 >analysis1 async

The input contains a stock ticker with 5 daily prices.

In one sentence, describe the trend and compute the simple range.

-->

<!--@2 <dataset_2 >analysis2 async

The input contains a stock ticker with 5 daily prices.

In one sentence, describe the trend and compute the simple range.

-->

```lua await=analysis1,analysis2
```

<!--@merge-->

<!--@1 <analysis1 <analysis2 >combined

Combine these two stock analyses into a unified two-sentence market summary.

-->

```lua <combined
```

```
local function read(v)
  local p = os.getenv("MSH_VAR_"..v)
  local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
  return t
end
print("=== Part 1: " .. read("dataset_1") .. " ==="); print("Analysis: " .. read("analysis1")); print()
print("=== Part 2: " .. read("dataset_2") .. " ==="); print("Analysis: " .. read("analysis2")); print()
print("=== Merged ==="); print(read("combined"))
```

AAPL:182,179,185,188,191

MSFT:374,371,378,382,385[split] dataset_1 = AAPL:182,179,185,188,191

[split] dataset_2 = MSFT:374,371,378,382,385

[async llm] Launched PID 60720 \u2192 var=analysis1 (model @1)

[async llm] Launched PID 60721 \u2192 var=analysis2 (model @2)

[async] await= barrier: waiting for vars: analysis1,analysis2

[async] Waiting for PID 60720 (var=analysis1)...

[async] PID 60720 done (var=analysis1)

AAPL shows an overall upward trend from 182 to 191 despite a minor dip to 179, with a simple range of 12 (191 - 179).

Both AAPL and MSFT demonstrated generally upward trends over the five-day period, with AAPL rising from 182 to 191 and MSFT climbing from 371 to 385. The stocks showed similar volatility with comparable ranges of 12 and 14 points respectively, suggesting a broadly positive market environment for major tech stocks.

=== Part 1: AAPL:182,179,185,188,191 ===

Analysis: AAPL shows an overall upward trend from 182 to 191 despite a minor dip to 179, with a simple range of 12 (191 - 179).

=== Part 2: MSFT:374,371,378,382,385 ===

Analysis: MSFT shows a generally upward trend over the five days, with a simple range of 14 (from 371 to 385).

=== Merged ===

Both AAPL and MSFT demonstrated generally upward trends over the five-day period, with AAPL rising from 182 to 191 and MSFT climbing from 371 to 385. The stocks showed similar volatility with comparable ranges of 12 and 14 points respectively, suggesting a broadly positive market environment for major tech stocks.

/home/igor > Sent to mshell (1014 bytes)

Received from GUI editor:

Pattern 17 CONFIG Node: Parameterized Pipeline

```config

topic=black holes

style=poetic and accessible

max_words=50

```

```lua >topic

io.write("black holes")

```

```lua >style

io.write("poetic and accessible")

```

```lua <style

io.write(io.open(os.getenv("MSH_VAR_style"),"r"):read("*all"))

```

<!--@1 <topic <style >explanation

The first input is a topic. The second input is a writing style.

Explain the topic in that style. Maximum 50 words.

-->

<!--@2 <explanation >keywords

Extract exactly 5 keywords as a comma-separated list. Nothing else.

-->

```lua <explanation <keywords >report
local function read(v)
  local p = os.getenv("MSH_VAR_"..v)
  local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
  return t
end
local lines = {
  "=== Workflow Report ===",
  "Topic : " .. read("topic"),
  "Style : " .. read("style"),
  "",
  "Explanation:",
  read("explanation"),
  "",
  "Keywords: " .. read("keywords"),
}
print(table.concat(lines, "\n"))
```

[config] topic=black holes

[config] style=poetic and accessible

[config] max_words=50

black holespoetic and accessiblepoetic and accessibleBlack holes are cosmic poets of gravity, writing verses in warped spacetime where light itself falls silent. These celestial sculptures carved from collapsed stars hold secrets in their event horizon embrace, whispering mathematics too beautiful for escape, turning the universe's grandest mysteries into elegant, invisible monuments.

black holes, gravity, spacetime, event horizon, mysteries

=== Workflow Report ===

Topic : black holes

Style : poetic and accessible

Explanation:

Black holes are cosmic poets of gravity, writing verses in warped spacetime where light itself falls silent. These celestial sculptures carved from collapsed stars hold secrets in their event horizon embrace, whispering mathematics too beautiful for escape, turning the universe's grandest mysteries into elegant, invisible monuments.

Keywords: black holes, gravity, spacetime, event horizon, mysteries

---

/home/igor > Sent to mshell (962 bytes)

Received from GUI editor:

Pattern 18 FOREACH + Async LLM: Parallel Lua Pattern Batch

```lua >patterns

io.write("observer\ncoroutine\nprototype")

```

<!--@foreach pattern in patterns-->

<!--@1 <pattern >explanation async

The input is a software design pattern. Explain it in one sentence for a beginner.

-->

<!--@2 <pattern >analogy async

The input is a software design pattern. Give one real-world analogy in one sentence.

-->

```lua await=explanation,analogy
```

```lua <explanation <analogy
local function read(v)
  local p = os.getenv("MSH_VAR_"..v)
  local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
  return t
end
local pp = os.getenv("MSH_VAR_pattern")
local fp = io.open(pp,"r"); local name = fp:read("*l"):gsub("%s+$",""); fp:close()
print("=== " .. name .. " ===")
print("Explanation : " .. read("explanation"))
print("Analogy     : " .. read("analogy"))
print()
```

<!--@end_foreach-->

```lua
print("=== Batch complete ===")
```

---

observer

coroutine

prototype[foreach] iter 1: pattern=observer

[async llm] Launched PID 60960 \u2192 var=explanation (model @1)

[async llm] Launched PID 60961 \u2192 var=analogy (model @2)

[async] await= barrier: waiting for vars: explanation,analogy

[async] Waiting for PID 60960 (var=explanation)...

[async] PID 60960 done (var=explanation)

The Observer pattern allows one object (the subject) to automatically notify multiple other objects (observers) when something important happens, like how a newspaper automatically delivers updates to all its subscribers when there's breaking news.

=== observer ===

Explanation : The Observer pattern allows one object (the subject) to automatically notify multiple other objects (observers) when something important happens, like how a newspaper automatically delivers updates to all its subscribers when there's breaking news.

Analogy : An observer pattern is like a news subscription service where, whenever a newspaper publishes a new edition, all subscribed readers automatically receive a copy without asking again.

[foreach] iter 2: pattern=coroutine

[async llm] Launched PID 61000 \u2192 var=explanation (model @1)

[async llm] Launched PID 61001 \u2192 var=analogy (model @2)

[async] await= barrier: waiting for vars: explanation,analogy

[async] Waiting for PID 61000 (var=explanation)...

[async] PID 61000 done (var=explanation)

A coroutine is a function that can pause its execution at certain points and later resume from exactly where it left off, allowing you to write code that appears to run simultaneously with other code without the complexity of traditional multithreading.

=== coroutine ===

Explanation : A coroutine is a function that can pause its execution at certain points and later resume from exactly where it left off, allowing you to write code that appears to run simultaneously with other code without the complexity of traditional multithreading.

Analogy : A coroutine is like two chefs sharing a single kitchen who take turns cooking parts of their dishes, pausing and resuming without ever leaving the room.

[foreach] iter 3: pattern=prototype

[async llm] Launched PID 61041 \u2192 var=explanation (model @1)

[async llm] Launched PID 61042 \u2192 var=analogy (model @2)

[async] await= barrier: waiting for vars: explanation,analogy

[async] Waiting for PID 61041 (var=explanation)...

[async] PID 61041 done (var=explanation)

The prototype pattern lets you create new objects by copying existing ones (prototypes) rather than building them from scratch, like using a cookie cutter to make identical cookies from dough.

=== prototype ===

Explanation : The prototype pattern lets you create new objects by copying existing ones (prototypes) rather than building them from scratch, like using a cookie cutter to make identical cookies from dough.

Analogy : A prototype pattern is like creating a custom key by duplicating an existing master key instead of cutting a brand new one from scratch each time.

=== Batch complete ===

/home/igor > Sent to mshell (1649 bytes)

Received from GUI editor:

Pattern 19 WHILE Quality Gate: Generate Lua Code Until Score >= 8

```lua >task
io.write("Write a haiku about the Lua programming language")
```

```lua >status
io.write("running")
```

```lua >iteration
io.write("0")
```

```lua >score
io.write("0")
```

```lua >poem
io.write("")
```

<!--@while status:running-->

```lua <iteration >iteration
local p = os.getenv("MSH_VAR_iteration")
local f = io.open(p,"r"); local v = tonumber(f:read("*l") or "0"); f:close()
local nv = v + 1
local fw = io.open(p,"w"); fw:write(tostring(nv)); fw:close()
print(nv)
```

<!--@1 <task >poem
Complete the creative writing task exactly as instructed. Reply with only the result.
-->

<!--@2 <poem >score
Rate this haiku 1-10 for creativity, imagery, and 5-7-5 form.
Reply with ONLY the integer score, nothing else.
-->

```lua <iteration <score <poem >status
local function read(v)
  local p = os.getenv("MSH_VAR_"..v)
  local f = io.open(p,"r"); local t = (f:read("*l") or ""):gsub("%s+$",""); f:close()
  return t
end
local function write(v, val)
  local p = os.getenv("MSH_VAR_"..v)
  local f = io.open(p,"w"); f:write(val); f:close()
```

end

local sc = read("score"):match("%d+") or "0"

print(string.format("[Iter %s] Score=%s", read("iteration"), sc))

print(read("poem")); print()

write("status", tonumber(sc) >= 8 and "done" or "running")

```

<!--@end_while-->

```lua <poem <score

local function read(v)

  local p = os.getenv("MSH_VAR_"..v)

  local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()

  return t

end

print("=== Accepted haiku (score=" .. read("score"):gsub("%s+","") .. ") ===")

print(read("poem"))

```

Write a haiku about the Lua programming languagerunning00[while] iteration 1 \u2014 condition met, executing body

1

Simple syntax flows

Tables hold both data, code\u2014

Lua scripts take flight

9

[Iter 1] Score=9

Simple syntax flows

[while] condition 'status:running' not met, exiting after 1 iter

=== Accepted haiku (score=9) ===
```

Simple syntax flows

Tables hold both data, code\u2014

Lua scripts take flight

/home/igor >

/home/igor > Sent to mshell (2255 bytes)

Received from GUI editor:

Pattern 20 SPLIT + Async + MERGE: Map-Reduce Pipeline

```lua >raw_text

io.write("Lua was designed at PUC-Rio in Brazil in 1993. It is one of the fastest scripting languages available. The language is embedded in millions of devices worldwide. Its coroutine system enables cooperative multitasking. Metatables give Lua a unique form of object-oriented programming.")

```

<!--@split raw_text into 3-->

```lua <raw_text >sent1

local p = os.getenv("MSH_VAR_raw_text")

local f = io.open(p,"r"); local text = f:read("*all"):gsub("%s+$",""); f:close()

local sents = {}

for s in text:gmatch("[^%.]+%.") do table.insert(sents, s:match("^%s*(.-)%s*$")) end

print(sents[1] or "")

```

```lua <raw_text >sent2

local p = os.getenv("MSH_VAR_raw_text")

local f = io.open(p,"r"); local text = f:read("*all"):gsub("%s+$",""); f:close()

local sents = {}

for s in text:gmatch("[^%.]+%.") do table.insert(sents, s:match("^%s*(.-)%s*$")) end
```

local chunk = {}

for i = 2, 3 do if sents[i] then table.insert(chunk, sents[i]) end end

print(table.concat(chunk, ". "))

```

```lua <raw_text >sent3

local p = os.getenv("MSH_VAR_raw_text")

local f = io.open(p,"r"); local text = f:read("*all"):gsub("%s+$",""); f:close()

local sents = {}

for s in text:gmatch("[^%.]+%.") do table.insert(sents, s:match("^%s*(.-)%s*$")) end

local chunk = {}

for i = 4, 5 do if sents[i] then table.insert(chunk, sents[i]) end end

print(table.concat(chunk, ". "))

```

<!--@1 <sent1 >analysis1 async

State the main concept in 3 words max.

-->

<!--@1 <sent2 >analysis2 async

State the main concept in 3 words max.

-->

<!--@1 <sent3 >analysis3 async

State the main concept in 3 words max.

-->

```lua await=analysis1,analysis2,analysis3

```

<!--@merge-->

<!--@2 <analysis1 <analysis2 <analysis3 >summary

Synthesize these three concept labels into one coherent theme sentence.
```

-->

```lua <analysis1 <analysis2 <analysis3 <summary
local function read(v)
  local p = os.getenv("MSH_VAR_"..v)
  local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
  return t
end
print("=== Map ===")
print("Chunk 1: " .. read("analysis1"))
print("Chunk 2: " .. read("analysis2"))
print("Chunk 3: " .. read("analysis3"))
print(); print("=== Reduce ==="); print(read("summary"))
```

Lua was designed at PUC-Rio in Brazil in 1993. It is one of the fastest scripting languages available. The language is embedded in millions of devices worldwide. Its coroutine system enables cooperative multitasking. Metatables give Lua a unique form of object-oriented programming.[split] raw_text_1 = Lua was designed at PUC-Rio in Brazil in 1993. It is one of the fastest scripting languages available. The language is embedded in millions of devices worldwide. Its coroutine system enables cooperative multitasking. Metatables give Lua a unique form of object-oriented programming.

Lua was designed at PUC-Rio in Brazil in 1993.

It is one of the fastest scripting languages available.. The language is embedded in millions of devices worldwide.

Its coroutine system enables cooperative multitasking.. Metatables give Lua a unique form of object-oriented programming.

[async llm] Launched PID 61537 \u2192 var=analysis1 (model @1)

[async llm] Launched PID 61538 \u2192 var=analysis2 (model @1)

[async llm] Launched PID 61540 \u2192 var=analysis3 (model @1)

[async] await= barrier: waiting for vars: analysis1,analysis2,analysis3

[async] Waiting for PID 61537 (var=analysis1)...

[async] PID 61537 done (var=analysis1)

Brazilian programming language

A fast, flexible Brazilian programming language designed for embedded and cooperative scripting.

=== Map ===

Chunk 1: Brazilian programming language

Chunk 2: Fast embedded scripting

Chunk 3: Flexible cooperative programming

=== Reduce ===

A fast, flexible Brazilian programming language designed for embedded and cooperative scripting.

/home/igor > Sent to mshell (1297 bytes)

Received from GUI editor:

Pattern 21 TRY/CATCH + LOOP: Resilient Lua Code Retry

```lua >task

io.write("Write Lua code that parses '{\"name\": \"Alice\"}' with pattern matching and prints the name field.")

```

```lua >result

io.write("fail")

```

```lua >last_error

io.write("none")

```

<!--@loop max=3 until=result:ok-->

<!--@1 <task <last_error >code

The first input is a coding task. The second is the previous error or "none".

Return ONLY Lua code, no fences, no explanation. Self-contained, runnable with lua5.4.

-->

```lua <code
local p = os.getenv("MSH_VAR_code")
local f = io.open(p,"r"); print("=== Generated Lua code ===");
print(f:read("*all"):gsub("%s+$","")); f:close()
```

<!--@try-->

```lua <code >result
local p = os.getenv("MSH_VAR_code")
local fn, load_err = loadfile(p)
if not fn then io.stderr:write(tostring(load_err).."\n"); os.exit(1) end
local ok_run, run_err = pcall(fn)
if not ok_run then io.stderr:write(tostring(run_err).."\n"); os.exit(1) end
print("ok")
```

<!--@catch >last_error-->

```lua >result
print("=== Error on this attempt: try_block_failed ===")
io.write("fail")
```

<!--@end_try-->

<!--@end_loop-->

```lua <result
local p = os.getenv("MSH_VAR_result")
local f = io.open(p,"r"); print("=== Final status: " .. f:read("*all"):gsub("%s+$","") .. " ===");
f:close()
```

```

---

Write Lua code that parses '{"name": "Alice"}' with pattern matching and prints the name field.failnone[loop] Starting loop: max=3 until=result:ok

local json_string = '{"name": "Alice"}'

local name = string.match(json_string, '"name"%s*:%s*"([^"]*)"')

print(name)

=== Generated Lua code ===

local json_string = '{"name": "Alice"}'

local name = string.match(json_string, '"name"%s*:%s*"([^"]*)"')

print(name)  0

[try] executing try block

Alice

ok

[try] try block succeeded

[loop] Exiting loop after 1 iteration(s). reason: until condition met

=== Final status: Alice

ok ===

---

/home/igor > Sent to mshell (1618 bytes)

Received from GUI editor:

---

# Pattern 22 Multi-Variable Output: Structured Field Extraction

```lua >input

io.write("Coroutines in Lua allow cooperative multitasking by letting functions pause and resume their execution, sharing a single thread without preemption.")

```

```lua <input

io.write(io.open(os.getenv("MSH_VAR_input"),"r"):read("*all"))

```

<!--@1 <input >raw_response

Respond in exactly this format (3 lines, no extra text):

SUMMARY: one sentence paraphrase

KEYWORDS: word1, word2, word3

AUDIENCE: one word \u2014 beginner / intermediate / expert

-->

```lua <raw_response >summary >keywords >audience

local p = os.getenv("MSH_VAR_raw_response")

local f = io.open(p,"r"); local text = f:read("*all"); f:close()

local function extract(label)

  return (text:match(label..":%s*(.-)%s*\n") or text:match(label..":%s*(.-)%s*$") or "n/a")

end

local summary  = extract("SUMMARY")

local keywords = extract("KEYWORDS")

local audience = extract("AUDIENCE")

local function wvar(v, val)

  local wp = os.getenv("MSH_VAR_"..v)

  local wf = io.open(wp,"w"); wf:write(val); wf:close()

end

wvar("summary", summary); wvar("keywords", keywords); wvar("audience", audience)

print("Summary  : " .. summary)

print("Keywords : " .. keywords)

print("Audience : " .. audience)

```

<!--@2 <summary <audience >adaptation
```

The first input is a summary. The second is a target audience level.

Rewrite the summary for that audience. One sentence only.

-->

```lua <adaptation

local p = os.getenv("MSH_VAR_adaptation")

local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()

print(); print("=== Adapted version ==="); print(t)

```

Coroutines in Lua allow cooperative multitasking by letting functions pause and resume their execution, sharing a single thread without preemption.Coroutines in Lua allow cooperative multitasking by letting functions pause and resume their execution, sharing a single thread without preemption.SUMMARY: Lua coroutines enable functions to voluntarily yield control and resume later, providing collaborative multitasking within a single thread.

KEYWORDS: cooperative, multitasking, yield

AUDIENCE: intermediate

Summary : Lua coroutines enable functions to voluntarily yield control and resume later, providing collaborative multitasking within a single thread.

Keywords : cooperative, multitasking, yield

Audience : intermediate

Lua coroutines let functions pause (yield) and later continue from the same point, enabling cooperative multitasking without using multiple operating system threads.

=== Adapted version ===

Lua coroutines let functions pause (yield) and later continue from the same point, enabling cooperative multitasking without using multiple operating system threads.

/home/igor > Sent to mshell (2301 bytes)

Received from GUI editor:

Pattern 23 CONFIG + WHILE + Multi-Model: Adaptive Pipeline

```config
subject=Lua coroutines
target_audience=high school student
quality_threshold=7
```

```lua >subject
io.write("Lua coroutines")
```

```lua >target_audience
io.write("high school student")
```

```lua >status
io.write("running")
```

```lua >iteration
io.write("0")
```

```lua >quality
io.write("0")
```

```lua >explanation
io.write("")
```

<!--@while status:running-->

```lua <iteration >iteration
local p = os.getenv("MSH_VAR_iteration")
```

local f = io.open(p,"r"); local v = tonumber(f:read("*l") or "0"); f:close()

local nv = v + 1

local fw = io.open(p,"w"); fw:write(tostring(nv)); fw:close()

print(nv)

```

<!--@1 <subject <target_audience >explanation

The first input is a subject. The second is the target audience.

Explain in exactly 3 sentences. No jargon. Be vivid and concrete.

You MUST respond \u2014 do not leave the response empty.

-->

<!--@2 <explanation <target_audience >quality

The first input is an explanation. The second is the target audience.

Rate clarity, accuracy, engagement on a scale 1-10.

Reply with ONLY the integer score, nothing else.

You MUST respond with a single integer \u2014 do not leave the response empty.

-->

```lua <iteration <quality >status

local function read(v)

  local p = os.getenv("MSH_VAR_"..v)

  local f = io.open(p,"r"); local t = (f:read("*l") or ""):gsub("%s+$",""); f:close()

  return t

end

local function write(v, val)

  local p = os.getenv("MSH_VAR_"..v)

  local f = io.open(p,"w"); f:write(val); f:close()

end

local q = read("quality"):match("%d+") or "0"

print(string.format("[Iter %s] Quality: %s", read("iteration"), q))

write("status", tonumber(q) >= 7 and "done" or "running")

```

<!--@end_while-->

<!--@1 <explanation >final_polish

Polish this text slightly for final publication. Keep exactly 3 sentences. No markdown.

You MUST respond \u2014 do not leave the response empty.

-->

```lua <final_polish <quality <iteration

local function read(v)

  local p = os.getenv("MSH_VAR_"..v)

  local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()

  return t

end

print(string.format("=== Final (score=%s, iters=%s) ===",

  read("quality"):gsub("%s+",""), read("iteration"):gsub("%s+","")))

print(read("final_polish"))

```

[config] subject=Lua coroutines

[config] target_audience=high school student

[config] quality_threshold=7

Lua coroutineshigh school studentrunning00[while] iteration 1 \u2014 condition met, executing body

1

Lua coroutines are like having a conversation where you can pause mid-sentence, let someone else talk, and then pick up exactly where you left off. Think of it as a function that can say "hold on, I'll be right back" and freeze in place, allowing other code to run, then
```

wake up and continue from that exact spot. This lets your program juggle multiple tasks smoothly, like switching between homework assignments without losing your place in any of them.

9

[Iter 1] Quality: 9

[while] condition 'status:running' not met, exiting after 1 iter

Lua coroutines are like having a conversation where you can pause mid-sentence, let someone else talk, and then pick up exactly where you left off. Think of it as a function that can say "hold on, I'll be right back" and freeze in place, allowing other code to run before waking up and continuing from that exact spot. This lets your program juggle multiple tasks smoothly, like switching between homework assignments without ever losing your place in any of them.

=== Final (score=9, iters=1) ===

Lua coroutines are like having a conversation where you can pause mid-sentence, let someone else talk, and then pick up exactly where you left off. Think of it as a function that can say "hold on, I'll be right back" and freeze in place, allowing other code to run before waking up and continuing from that exact spot. This lets your program juggle multiple tasks smoothly, like switching between homework assignments without ever losing your place in any of them.

/home/igor > Sent to mshell (1272 bytes)

Received from GUI editor:

Pattern 24 FOREACH + TRY/CATCH: Fault-Tolerant Batch Processing

```lua >items

io.write('x=1;y=nil\nx= broken !!\nx=nil;y=42')

```

<!--@foreach item in items-->

<!--@try-->

```lua <item >parsed

local p = os.getenv("MSH_VAR_item")

local f = io.open(p,"r"); local raw = f:read("*l"):gsub("%s+$",""); f:close()
```

````
local fn, err = load("local " .. raw .. "; return {" .. raw .. "}")
if not fn then io.stderr:write("Parse error: "..tostring(err).."\n"); os.exit(1) end
local ok, result = pcall(fn)
if not ok then io.stderr:write("Eval error: "..tostring(result).."\n"); os.exit(1) end
local parts = {}
for k, v in pairs(result) do
  if v ~= nil then table.insert(parts, tostring(k).."="..tostring(v)) end
end
print("Parsed OK: {" .. table.concat(parts, ", ") .. "}")
```

<!--@1 <parsed >insight
The input is a parsed data object description. In one sentence, describe what data it contains.
-->

```lua <insight
local p = os.getenv("MSH_VAR_insight")
local f = io.open(p,"r"); local t = f:read("*all"):gsub("%s+$",""); f:close()
print("[OK] " .. t)
```

<!--@catch >parse_error-->

```lua
print("[ERR] Parse failed: try_block_failed")
```

<!--@end_try-->

<!--@end_foreach-->

```lua
````

print("=== Batch complete. Errors were isolated, pipeline never stopped. ===")

```

---

x=1;y=nil

x= broken !!

x=nil;y=42[foreach] iter 1: item=x=1;y=nil

[try] executing try block

Parsed OK: {x=1}

The parsed data contains a single key-value pair with a variable named "x" that has the numeric value 1.

[OK] The parsed data contains a single key-value pair with a variable named "x" that has the numeric value 1.

[try] try block succeeded

[foreach] iter 2: item=x= broken !!

[try] executing try block

Parse error: [string "local x= broken !!; return {x= broken !!}"]:1: unexpected symbol near '!'

The parsed data contains a table with one field "x" that holds the integer value 1.

[try] try block failed, executing catch block

[ERR] Parse failed: try_block_failed

[foreach] iter 3: item=x=nil;y=42

[try] executing try block

Parsed OK: {y=42}

The parsed data contains a table with one field "y" that holds the integer value 42.

[OK] The parsed data contains a table with one field "y" that holds the integer value 42.

[try] try block succeeded

=== Batch complete. Errors were isolated, pipeline never stopped. ===

---

References:
```

Lua language Patterns for mshell Workflow — Complete Reference Guide (P1–P24) Pure Lua Edition — Art2Dec SoftLab (Non-profitable SoftLab), 2026 Created by Igor Lukyanov, Art2Dec SoftLab Based on the original mshell Workflow Patterns Reference Guides Part I & Part II

Resources: - Common examples Part I (P1–P12):
https://www.appservgrid.com/paw92/index.php/2026/02/26/mshell-workflow-patterns-reference-guide-part-i-p1-p13/

Resources: - Common examples Part II (P13–P24):
https://www.appservgrid.com/paw92/index.php/2026/03/11/mshell-workflow-patterns-reference-guide-part-ii-p13-p24/

- mshell v1.4.1 cheatsheet:
https://www.appservgrid.com/paw92/index.php/2026/02/04/mshell-v-1-4-1-cheatsheet-january-26th-2026/