Pure mshell language Patterns for mshell Workflow — Complete Reference Guide (P1–P24)

Pure mshell language Edition — Art2Dec SoftLab, March 17th 2026

What is mshell?

mshell is a polyglot UNIX shell environment for AI and mathematics that integrates multiple programming languages with AI model capabilities into a single unified execution pipeline. This guide presents all 24 workflow patterns using **mshell** as the primary language.

---

mshell Command Reference

Command	Description
`print "text"`	Output to stdout (captured as `>outvar`)
`print $var`	Print variable value
`set x = value`	Assign variable
`set x = eval expr`	Assign computed expression
`readfile $MSH_VAR_name`	Read context variable file
`writefile $MSH_VAR_name "val"`	Write to context variable file (unreliable in WHILE)
`ollama1 "prompt $var"`	Call model @1 inline — captures to stdout
`ollama2 "prompt $var"`	Call model @2 inline
`ollama1exec "task"`	LLM @1 generates+executes — MUST have `>outvar`
`exec readfile $MSH_VAR_code`	Execute mshell code from variable file

Variable System

Syntax	Effect
`` ```mshell >varname``` ````	Capture stdout → variable
`` ```mshell <varname``` ````	Inject variable as `$varname`
`` ```mshell <v1 <v2 >out``` ````	Multiple inputs, one output
`` ```mshell >a >b``` ````	Multiple outputs: use `writefile`

Syntax	Effect
	`$MSH_VAR_*`

Reading variables reliably: - `$varname` — works for short single-line values set by mshell/ollama blocks - `readfile $MSH_VAR_name` — works for mshell-written vars, NOT for LLM directive vars - Python `open(os.environ['MSH_VAR_name']).read()` — works for ALL vars including LLM

Control Flow

Syntax	Semantics
`<!--@while var:value-->` … `<!--@end_while-->`	Loop while var == value
`<!--@foreach item in listvar-->` … `<!--@end_foreach-->`	Iterate over lines
`<!--@loop max=N until=var:value-->` … `<!--@end_loop-->`	Bounded loop, checks last stdout line
`<!--@try-->` … `<!--@catch >errvar-->` … `<!--@end_try-->`	Error isolation
`<!--@split var into N-->`	Creates var_1, var_2, …
`<!--@merge-->`	Visual reduce marker
`if=varname:value` on any block	Conditional execution

Part I — Foundation Patterns (P1–P12)

Pattern 1 Linear Pipeline: Chain of MShell Transformations

What it does: A seed number flows through a pure mshell pipeline: square 192 prime check 192 format 192 report.

Key concept: Variables flow top-to-bottom. Use `$varname` directly in blocks that declare `<varname`. Use `ollama1` for computation that mshell cannot do natively.

Flow diagram

```
mshell >seed \u2193 mshell <seed >squared    (ollama1: compute square) \u2193 mshell <squared >verdict (ollama1: prime check) \u2193 mshell <seed <squared <verdict >report \u2193 mshell <report
```

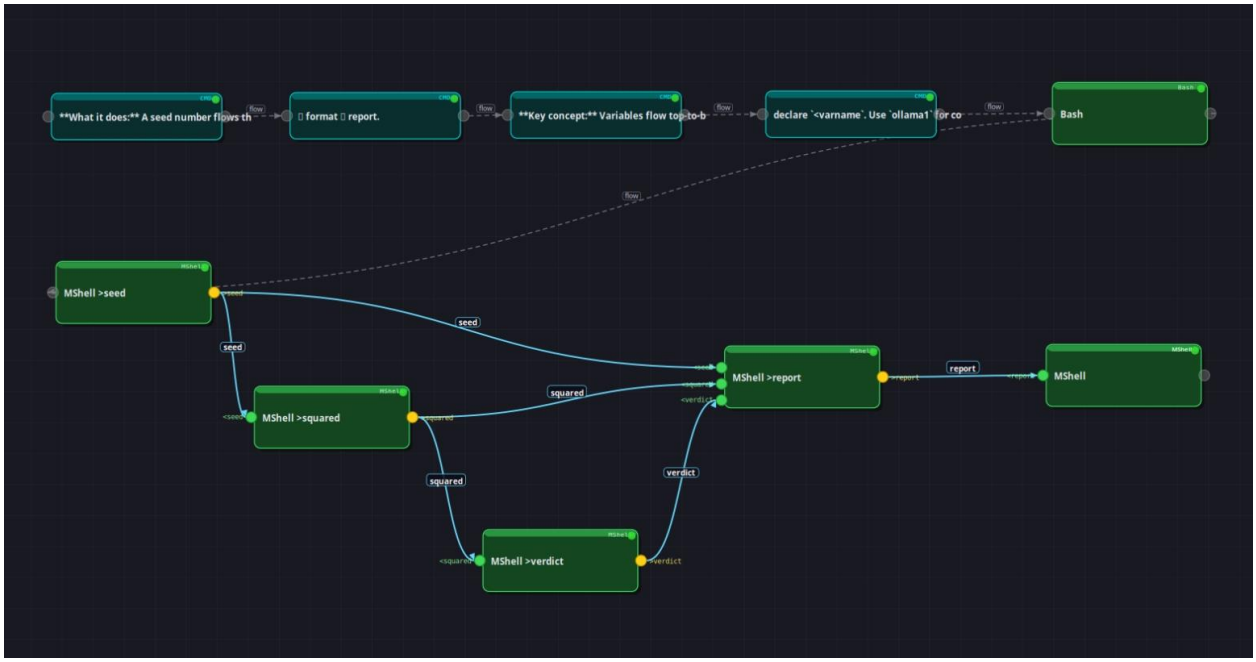
Code

```
mshell >seed print 7
```

```
mshell <seed >squared ollama1 "Compute $seed * $seed. Reply with ONLY the integer result, nothing else."
```

```
mshell <squared >verdict ollama1 "Is $squared a prime number? Reply with
```

```
exactly one word: PRIME or COMPOSITE."
mshell <seed <squared <verdict >report print "Number: $seed | Square:
$squared | Classification: $verdict"
mshell <report print "=== Pipeline Result ===" print $report
```



Pattern 2 LLM in the Middle: AI as a Transformation Layer

What it does: mshell generates CSV sensor data. LLM @1 analyzes for anomalies. mshell reads the report and prints a meta-summary.

Key concept: LLM directive injects variable contents into the prompt and stores the response in an output variable.

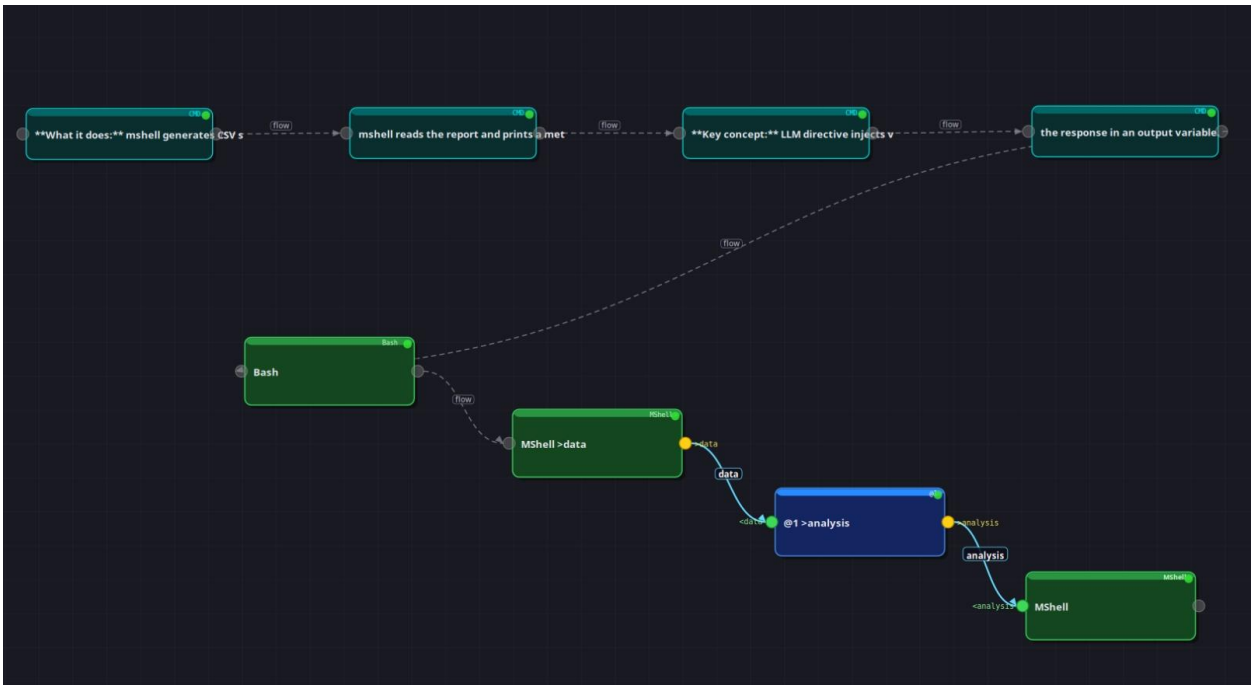
Flow diagram

```
mshell >data \u2193 @1 <data >analysis \u2193 mshell <analysis
```

Code

```
mshell >data print "sensor_id,timestamp,temp_c" print "S01,08:00,22.1" print
"S01,08:05,22.4" print "S01,08:10,41.7" print "S01,08:15,22.0" print
"S01,08:20,22.3"


mshell <analysis print "=== Anomaly Report ===" print $analysis
```



Pattern 3 Fan-Out: One Source, Many Consumers

What it does: A single variable is consumed by two mshell blocks and one LLM independently. Each reads the same unchanged file.

Key concept: Multiple blocks can reference `<varname`. All consumers read the same file.

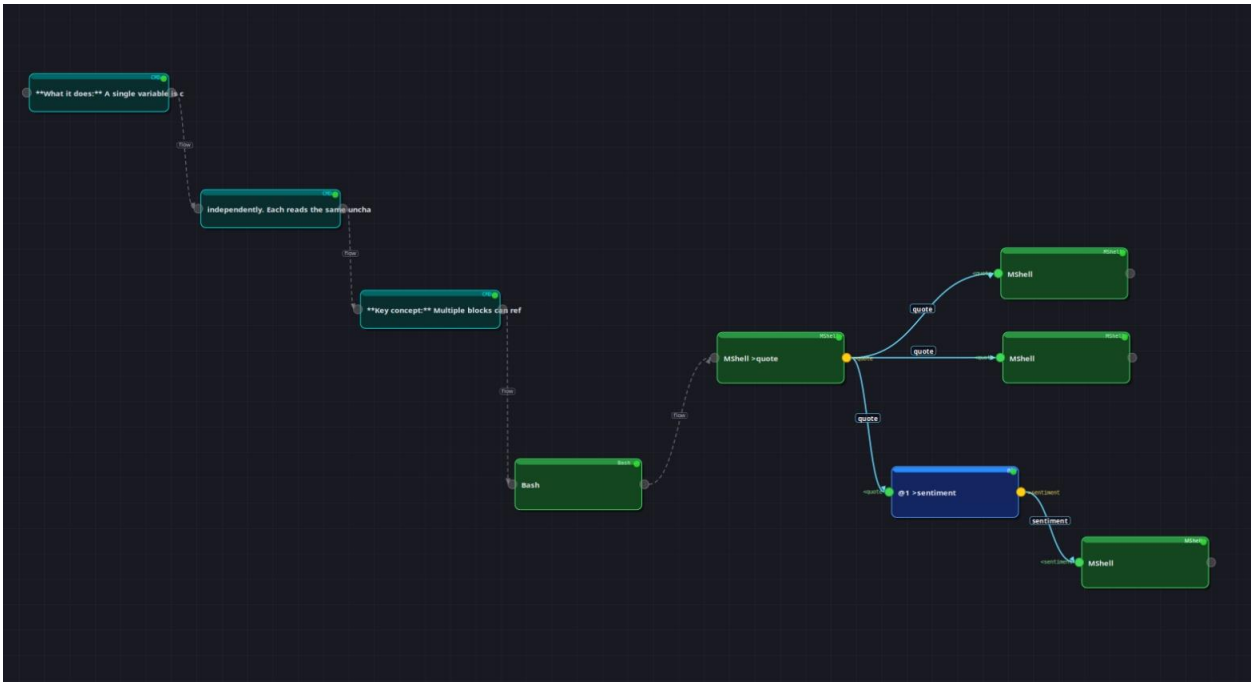
Flow diagram

```
mshell >quote \u2193         \u2193              \u2193 mshell      mshell
@1 <quote <quote      <quote         >sentiment
```

Code

```
mshell >quote print "The unexamined life is not worth living."
mshell <quote print "Word count: $(len $(split($quote, ' ')))" print "Quote:
$quote"
mshell <quote ollama1 "Reverse the word order of this sentence: $quote. Reply
with ONLY the reversed sentence."


mshell <sentiment print "=== Sentiment Analysis ===" print $sentiment
```



Pattern 4 LLM Code Generation - Execute via Variable

What it does: `ollama1exec` asks LLM @1 to generate and execute mshell code. Result is captured into `>result` and displayed via `readfile`.

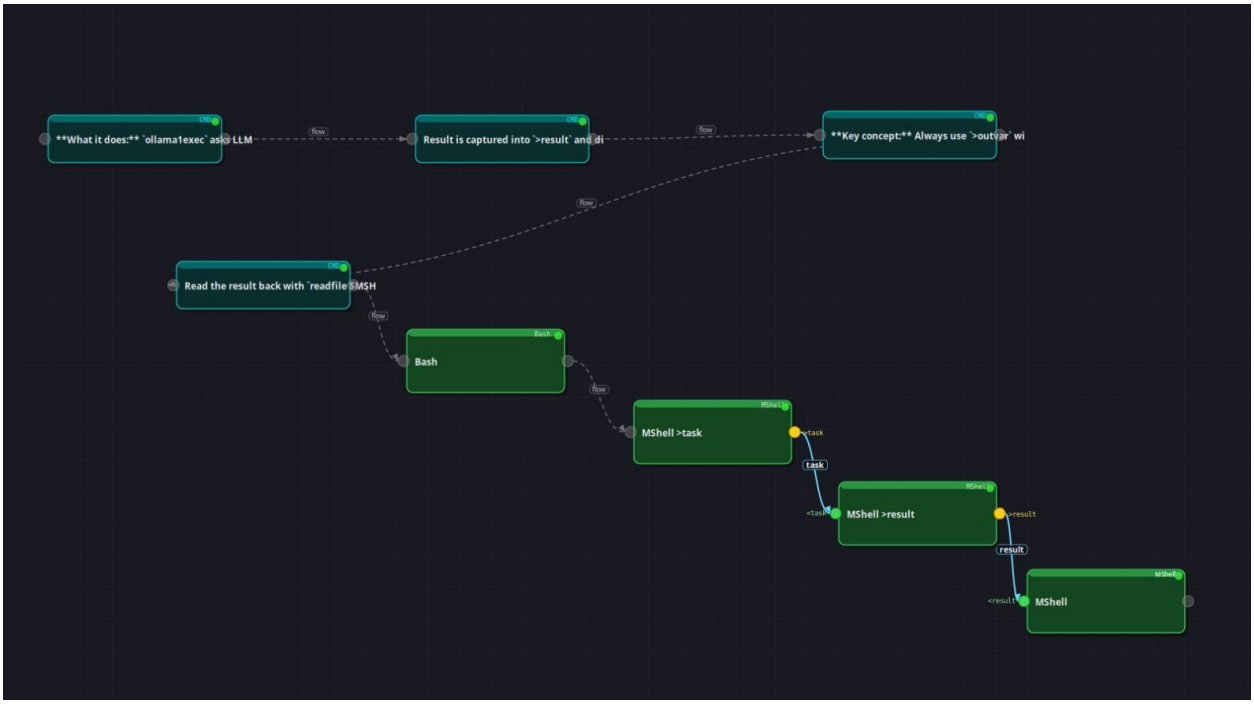
Key concept: Always use `>outvar` with `ollama1exec` to prevent mshell from crashing. Read the result back with `readfile $MSH_VAR_result` 014 not `$result`.

Flow diagram

```
mshell >task \u2193 mshell <task >result   (ollama1exec: generate + execute)
\u2193 mshell <result         (readfile to display)
```

Code

```
mshell >task print "Generate the Collatz sequence starting from 27. Print
each number on its own line until you reach 1."
mshell <task >result ollama1exec "Generate and execute mshell code for this
task: $task. Use mshell syntax: set, while, if, eval, print."
mshell <result set text = readfile $MSH_VAR_result print "=== Result ==="
print $text
```



Pattern 5 Two-LLM Review Chain: Generate - Review - Improve

What it does: @1 generates a riddle. @2 critiques it. @1 receives both and produces an improved version.

Key concept: Multiple `<invar` auto-labeled as `[varname]:` in the prompt.

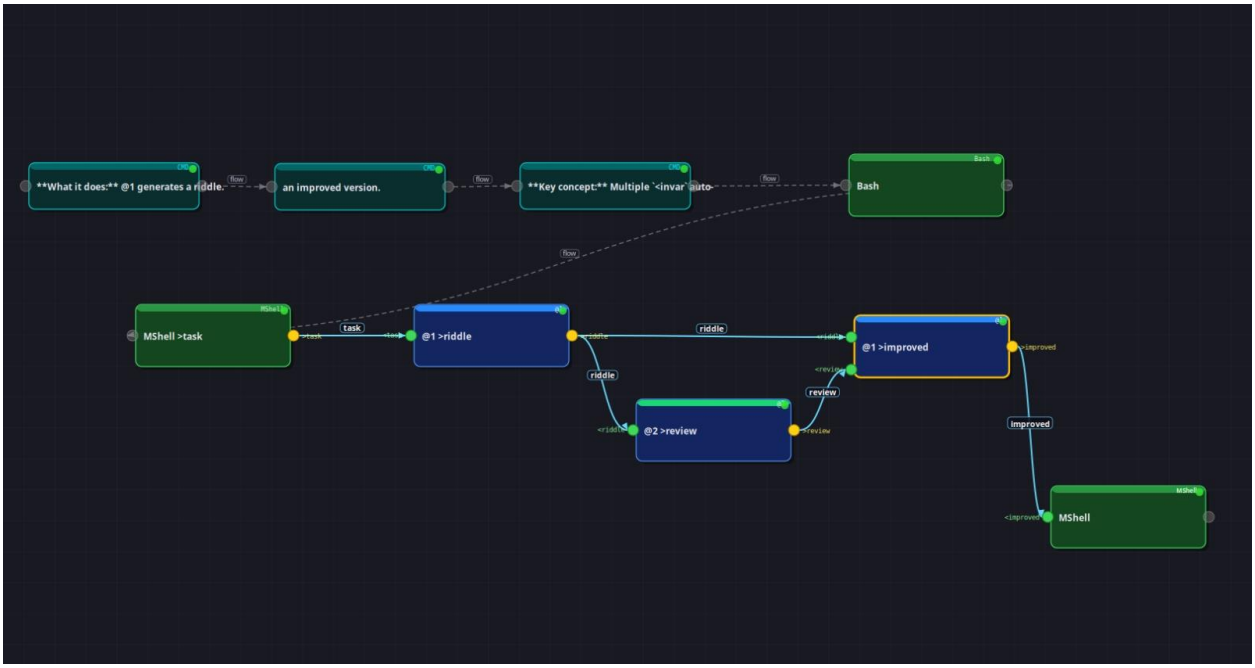
Flow diagram

```
mshell >task \u2193 @1 <task >riddle \u2193 @2 <riddle >review \u2193 @1 <riddle <review >improved \u2193 mshell <improved
```

Code

```
mshell >task print "Write a short riddle about time. Include the answer at the end after 'Answer:'. Make it 3-4 lines."
```

```
mshell <improved print "=== Final Improved Riddle ===" print $improved
```



Pattern 6 Parallel 3-Model Query: Three Philosophical Traditions

What it does: The same question sent to all three models. Each adopts a distinct philosophical tradition. mshell collects and displays all three.

Key concept: Three LLM blocks with different model numbers all read `<question` and write to distinct output variables.

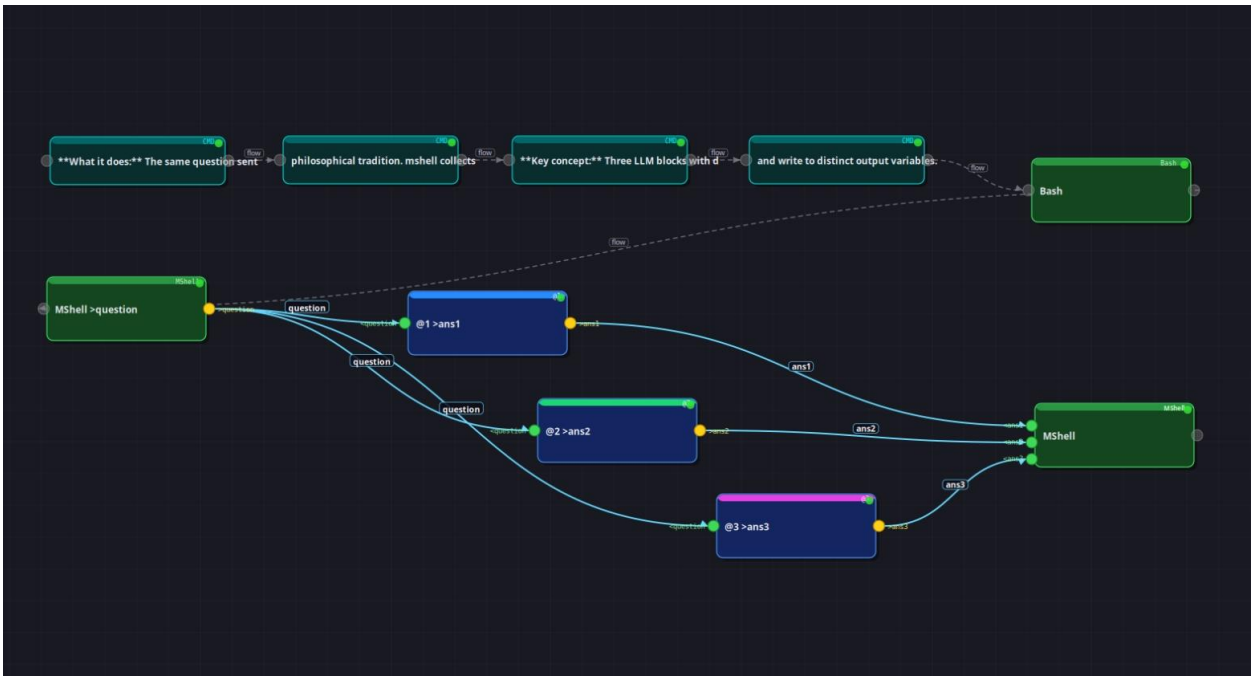
Flow diagram

```
mshell >question \u2193 @1 <question >ans1   @2 <question >ans2   @3
<question >ans3 \u2193 mshell <ans1 <ans2 <ans3
```

Code

```
mshell >question print "What should a person do when faced with an outcome
they cannot control?"
```

```
mshell <ans1 <ans2 <ans3 print "[Stoicism]       $ans1" print "[Zen]
$ans2" print "[Existentialism] $ans3"
```



Pattern 7 Evaluator-Optimizer Loop: Generate Until Accepted

What it does: @1 generates a mnemonic for the 8 planets. @2 evaluates strictly: `ACCEPTED` or `REJECTED: reason`. Loop repeats max 3 times until accepted.

Key concept: `<!--@loop max=N until=verdict:ACCEPTED-->` exits early on acceptance.

Flow diagram

```
mshell >task \u2193 [LOOP max=3 until=verdict:ACCEPTED] @1 <task <verdict
>mnemonic mshell <mnemonic @2 <mnemonic >verdict mshell <verdict [END_LOOP]
\u2193 mshell <mnemonic
```

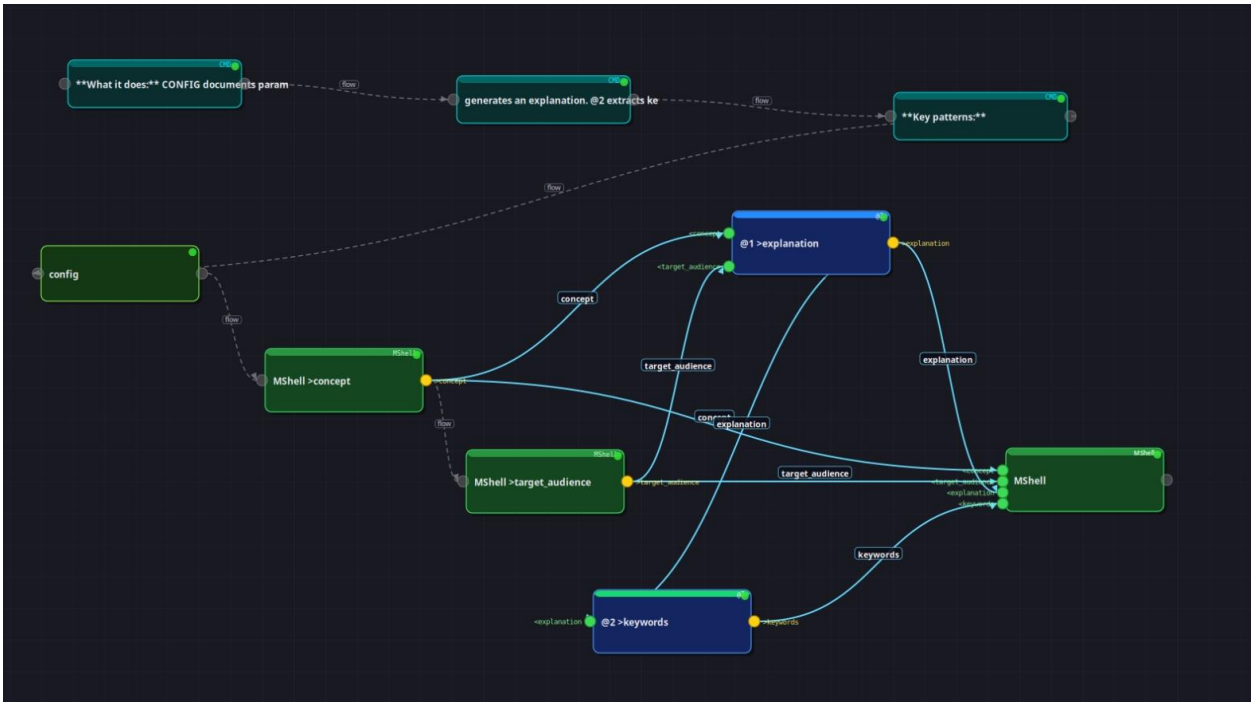
Code

```
mshell >task print "Create a mnemonic sentence where each word starts with
the first letter of a planet in order: Mercury Venus Earth Mars Jupiter
Saturn Uranus Neptune."


mshell <mnemonic print "--- Attempt: $mnemonic"


mshell <verdict print "--- Verdict: $verdict"


mshell <mnemonic print "=== Accepted: $mnemonic ==="
```



Pattern 8 Multi-Model Pipeline: Compute - Analyze - Summarize

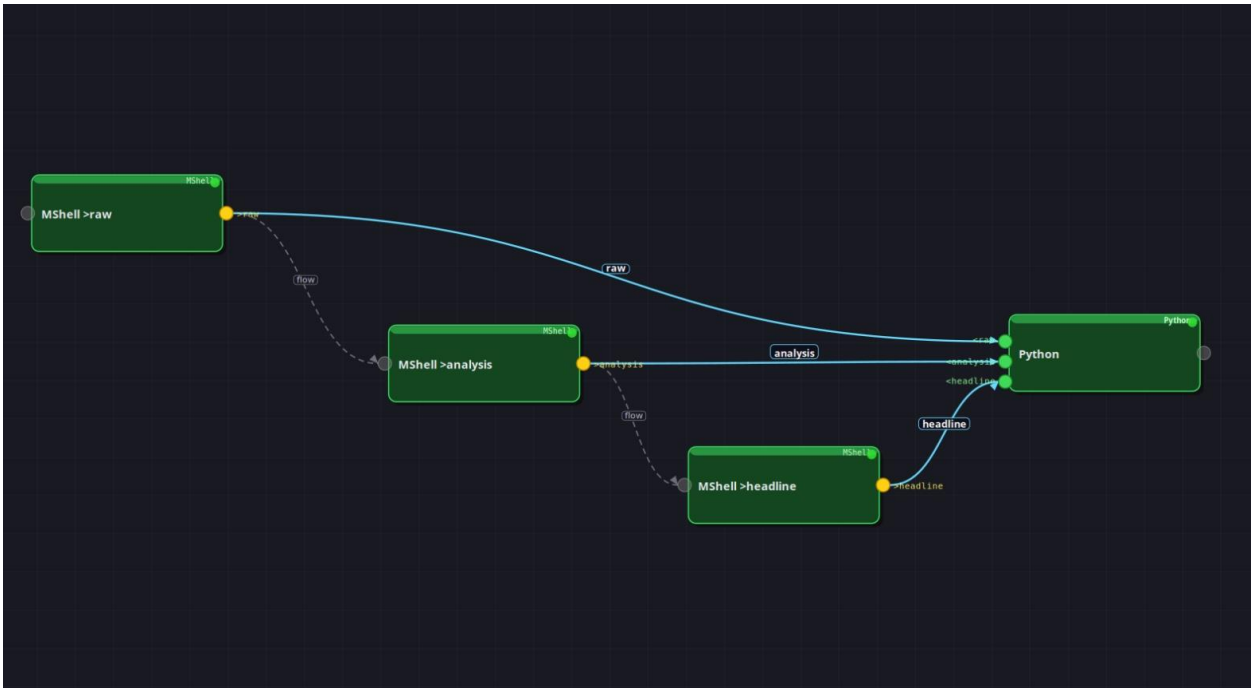
Code

```
mshell >raw ollama1exec "Print the first 10 Fibonacci numbers separated by spaces on one line. Print ONLY the numbers, nothing else: 1 1 2 3 5 8 13 21 34 55"

mshell >analysis ollama1exec "In one sentence, describe what the first 10 Fibonacci numbers (1 1 2 3 5 8 13 21 34 55) reveal about exponential growth and the golden ratio."

mshell >headline ollama1exec "Give a 5-word tweet-style headline about Fibonacci numbers and the golden ratio. Nothing else."

python <raw <analysis <headline import os raw      = open(os.environ['MSH_VAR_raw']).read().strip() analysis = open(os.environ['MSH_VAR_analysis']).read().strip() headline = open(os.environ['MSH_VAR_headline']).read().strip() print(f"[Data] {raw}") print(f"[Analysis] {analysis}") print(f"[Headline] {headline}")
```



Pattern 9 Routing: LLM Classifies - Conditional Branch Executes

What it does: @1 classifies a query into SCIENCE / HISTORY / PHILOSOPHY. Only the matching branch executes. All three test cases shown.

Key concept: `if=varname:expected_value` makes any block conditional.

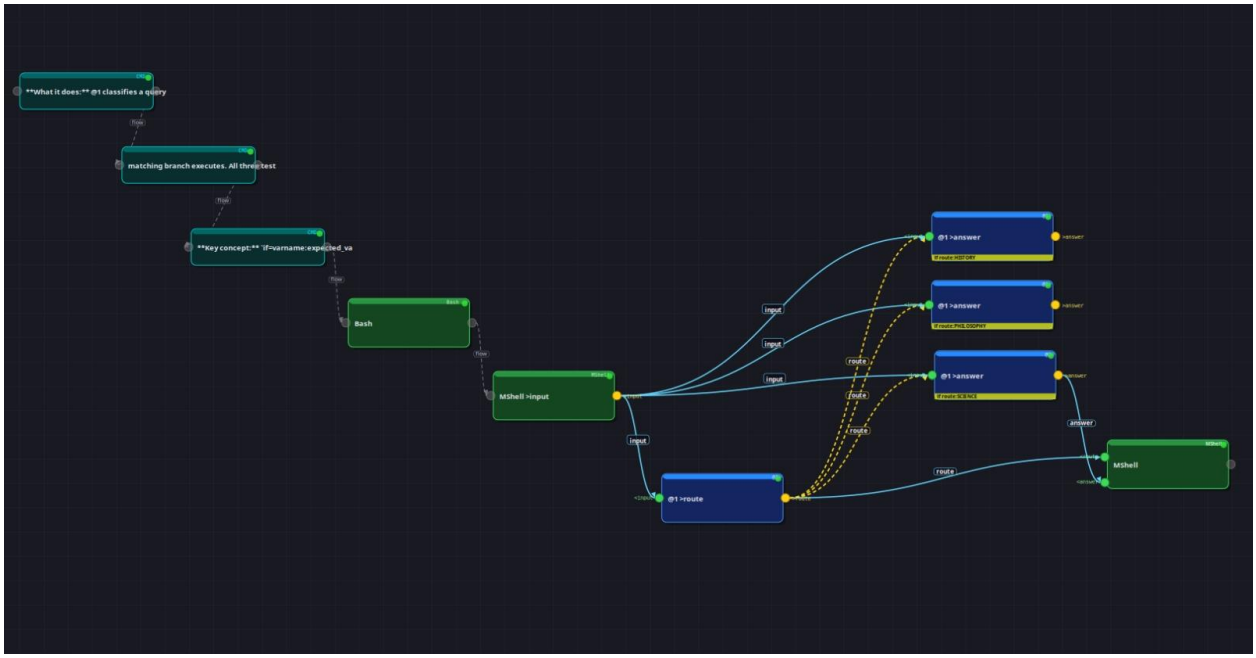
Flow diagram

```
mshell >input \u2193 @1 <input >route \u2193 @1 <input if=route:SCIENCE
>answer @1 <input if=route:HISTORY >answer @1 <input if=route:PHILOSOPHY
>answer \u2193 mshell <route <answer
```

Code 014 Test 1: SCIENCE

```
mshell >input print "Why does water expand when it freezes?"
```

```
mshell <route <answer print "Route: $route" print "Answer: $answer"
```



Pattern 10 Full Pipeline: All Patterns Combined

What it does: mshell computes perfect squares. Two LLMs run async (analyst + poet). Await barrier synchronizes. Third LLM synthesizes. Final mshell formats the output.

Key concept: Patterns compose naturally 014 every block reads named variables.

Flow diagram

```
mshell >raw_data       (perfect squares) \u2193 mshell <raw_data >stats
\u2193 @1 <stats >analysis async \u2500\u2510 @2 <raw_data >poem async
\u2500\u2524 mshell await=analysis,poem \u2500\u2518 \u2193 @1 <analysis
<poem >combined \u2193 mshell <combined
```

Code

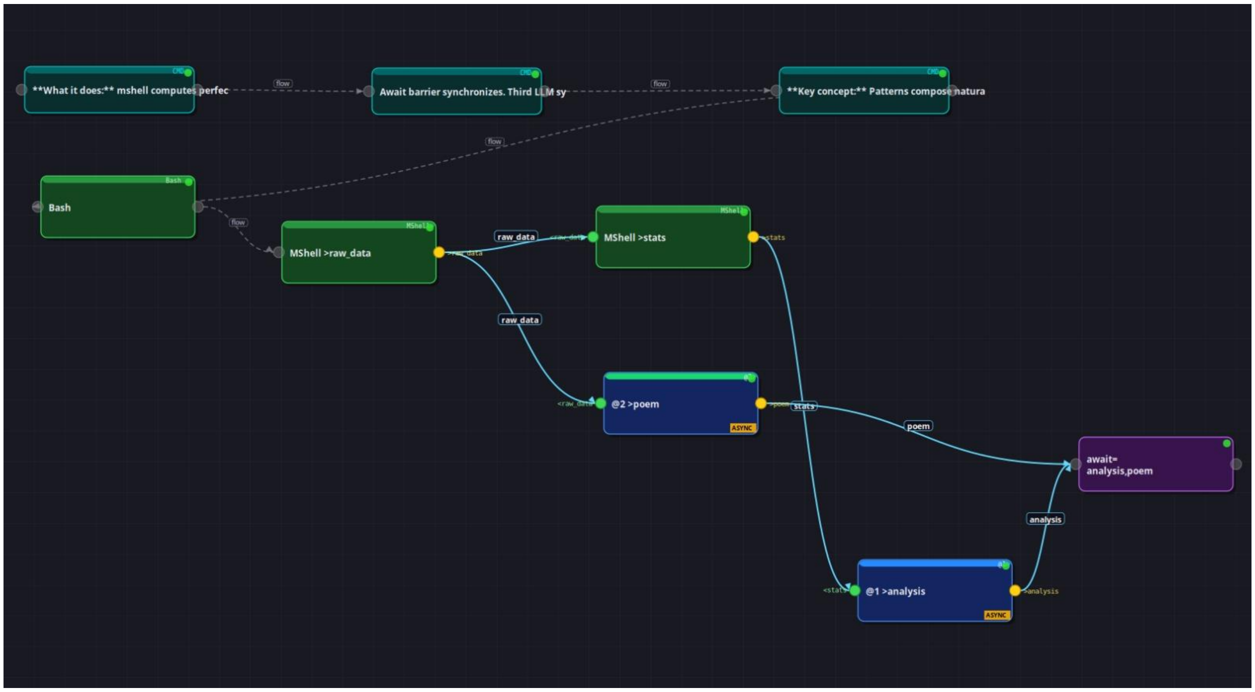
```
mshell >raw_data set out = "" set i = 1 while eval $i <= 8 set sq = eval $i *
$i if eval $i == 1 set out = $sq else set out = "$out,$sq" end set i = eval
$i + 1 end print $out
```

```
mshell <raw_data >stats ollama1 "The input is a CSV of perfect squares:
$raw_data. Compute min, max, and mean. Reply in one line: min=X max=Y mean=Z"
```

```
mshell await=analysis,poem
```

```
mshell <combined print
"\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550" print $combined print
```

```
"\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550"
```



Pattern 11 MShell Native AI Node: ollama Commands in mshell Blocks

What it does: Uses `ollama1`/`ollama2` commands inside mshell blocks. Topic and style are set via mshell. First block calls ollama1 inline. Second passes result to ollama2.

Key concept: `ollama1 "prompt $varname"` calls model @1 inline; `$varname` from `<invar` is available directly in the prompt string.

Flow diagram

```
mshell >topic mshell >style \u2193 mshell <topic <style >explanation
(ollama1 inline) \u2193 mshell <explanation >keywords       (ollama2 inline)
\u2193 mshell <explanation <keywords
```

Code

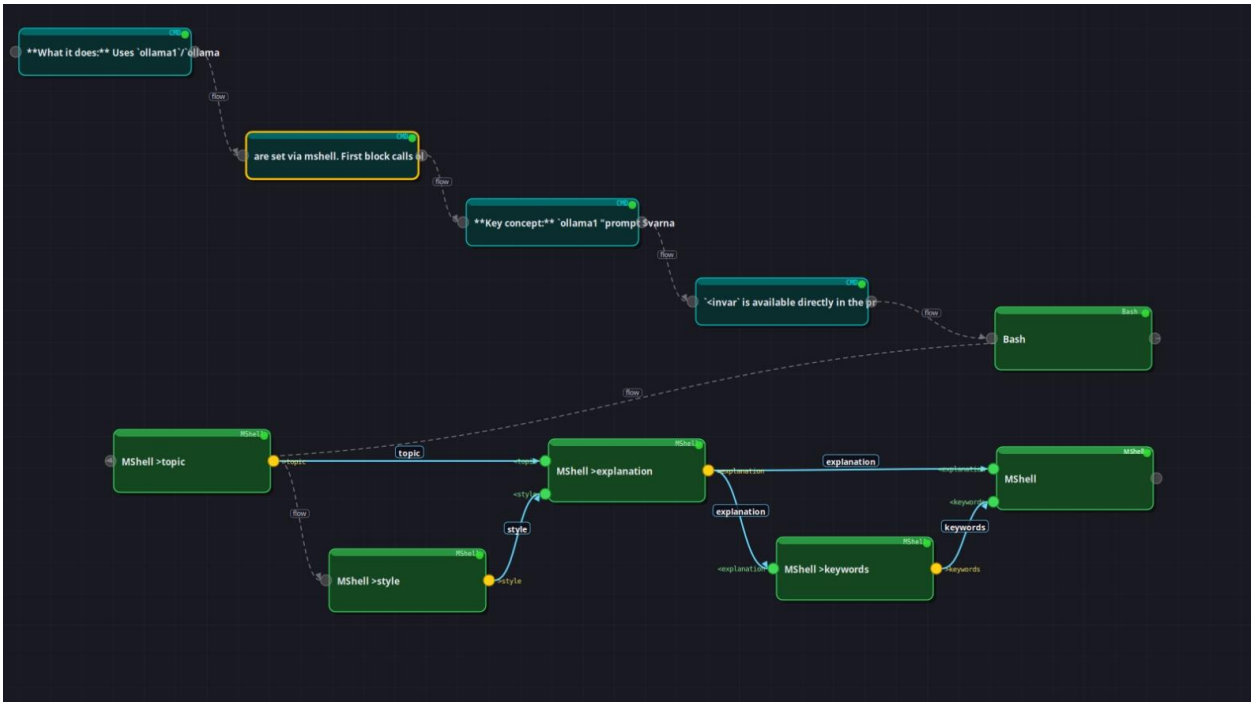
```
mshell >topic print "quantum superposition"

mshell >style print "vivid analogy for a 12-year-old"

mshell <topic <style >explanation ollama1 "Explain $topic using $style. One
sentence only."

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

mshell <explanation <keywords print "Explanation: $explanation" print
"Keywords   : $keywords"
```



Pattern 12 Async Parallel 3 Models + Await Barrier + Synthesis

What it does: Three LLM models answer a question asynchronously (neuroscience / computation / phenomenology). Await barrier synchronizes. Fourth LLM synthesizes.

Key concept: `async` launches in background. `mshell await=v1,v2,v3` blocks until all variables are written. Total time = slowest model, not the sum.

Flow diagram

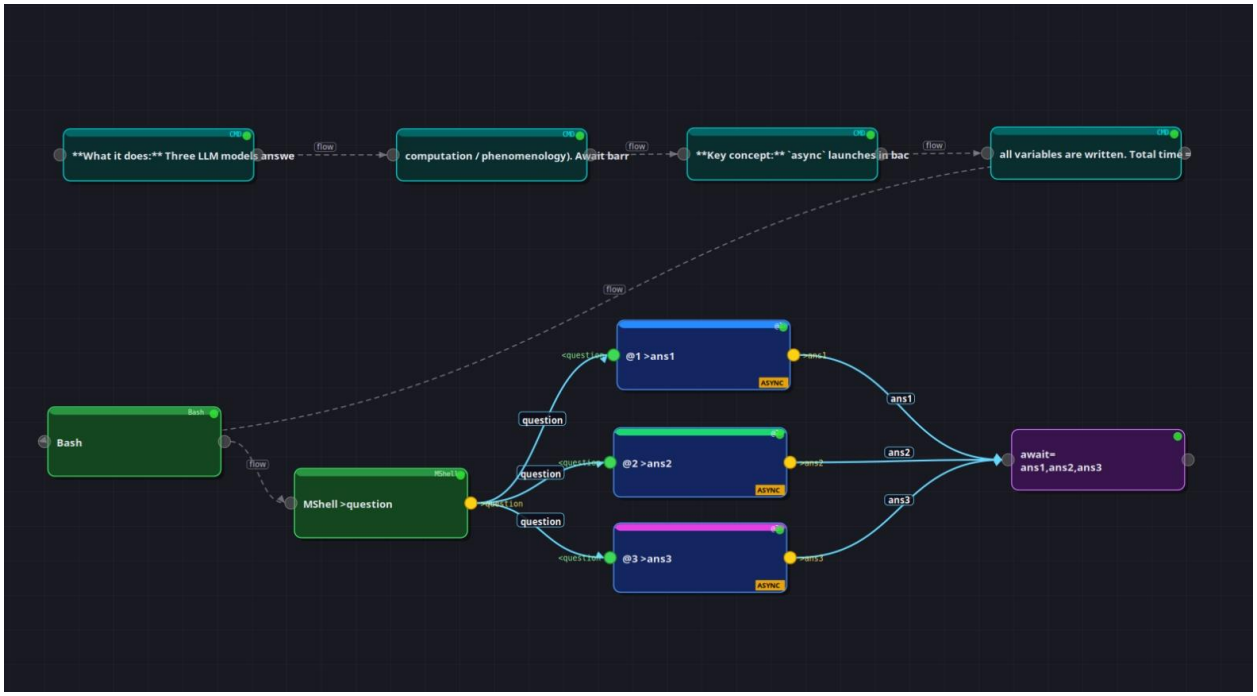
```
mshell >question \u2193 @1 <question >ans1 async \u2500\u2510 @2 <question
>ans2 async \u2500\u2524 @3 <question >ans3 async \u2500\u2518 mshell
await=ans1,ans2,ans3 \u2193 @1 <ans1 <ans2 <ans3 >final \u2193 mshell <final
```

Code

```
mshell >question print "What is consciousness and can a machine ever truly
have it?"
```

```
mshell await=ans1,ans2,ans3
```

```
mshell <ans1 <ans2 <ans3 <final print "[Neuroscience]  $ans1" print
"[Computation]   $ans2" print "[Phenomenology] $ans3" print "" print "===
Synthesis ===" print $final
```



Part II — Advanced Patterns (P13–P24)

Pattern 13 Iterative Counter with LLM Commentary

What it does: counter=0, status=running. WHILE loop. Python block increments counter and writes status via MSH_VAR_* files directly. @1 gives math fact per number.

Key concept: Use Python for counter increment in mshell WHILE loops 014 Python can write to MSH_VAR_* files reliably where mshell writefile fails.

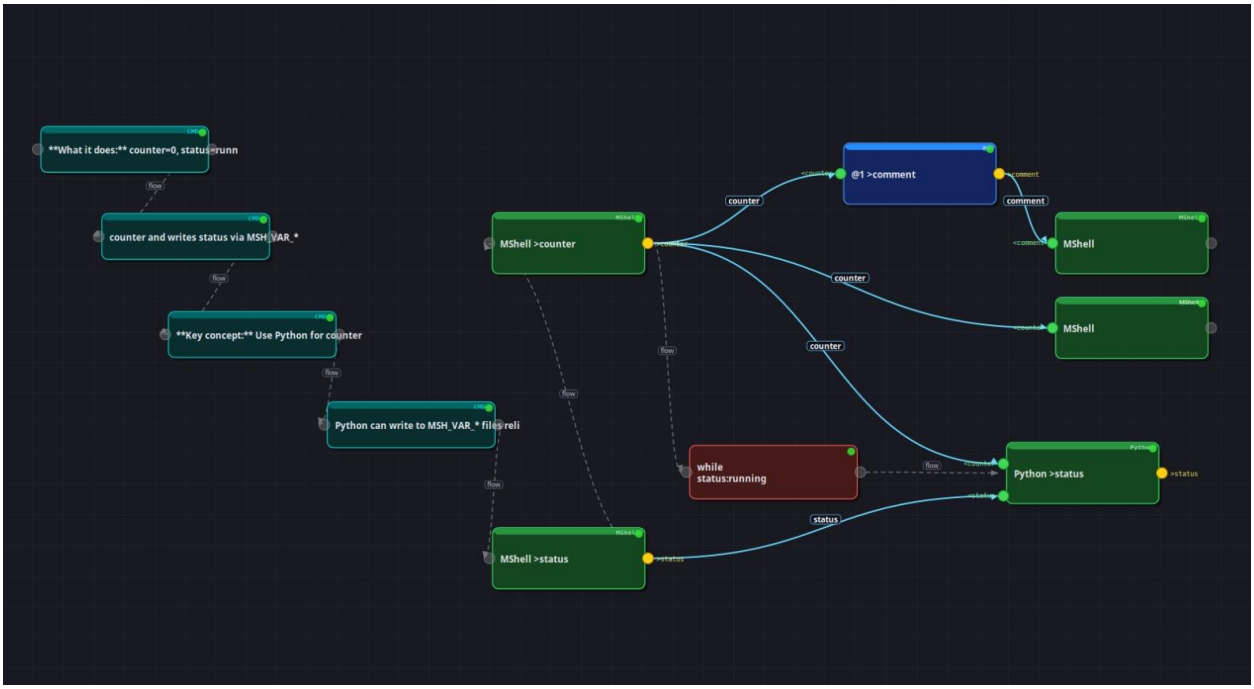
Code

```
mshell >status print "running"
mshell >counter print "0"
```

```python <counter counter >status import os
counter_path = os.environ['MSH_VAR_counter'] status_path = os.environ['MSH_VAR_status']
val = int(open(counter_path).read().strip() or '0') newval = val + 1
open(counter_path, 'w').write(str(newval)) open(status_path, 'w').write('done' if newval >= 4 else 'running')
print(newval) ```

```
mshell <comment print "[iter $counter] $comment"

mshell <counter print "=== WHILE complete. Final counter = $counter ==="
```
```



Pattern 14 FOREACH: LLM Processes Each Item in a List

What it does: A newline-separated list of world capitals. FOREACH iterates line by line. @1 generates one cultural fact per city. mshell prints each result.

Key patterns: - Use separate `print` per item 014 mshell does NOT interpret `\n` in strings. - The iterator variable is set automatically per iteration.

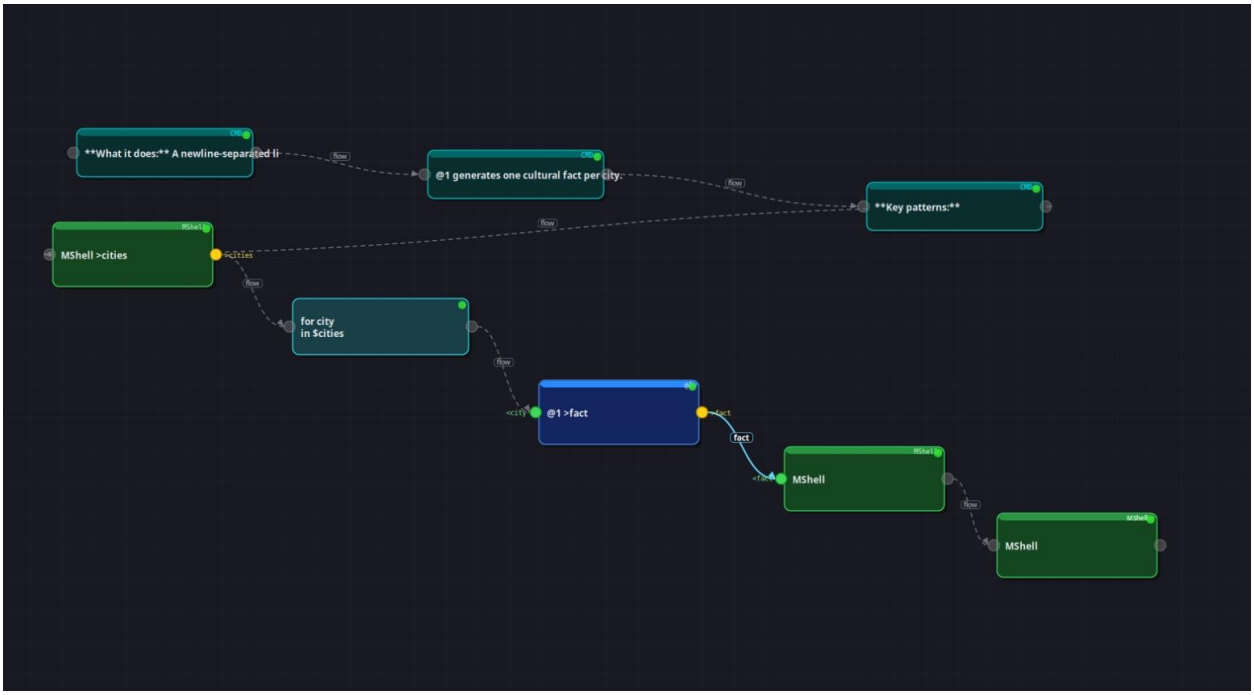
Code

```
mshell >cities print "Tokyo" print "Cairo" print "Buenos Aires" print
"Reykjavik"



mshell <fact print "--- $city ---" print $fact print ""


mshell print "=== All capitals explored ==="
```



Pattern 15 TRY/CATCH: Safe Execution with Error Capture

What it does: TRY block attempts evaluation of an invalid expression. On failure, CATCH prints a hardcoded message. Safe fallback computes a valid expression.

Key patterns: - CATCH does NOT use `<errvar` 014 print `"try_block_failed"` literally. - Pipeline continues normally after `<!--@end_try-->`.

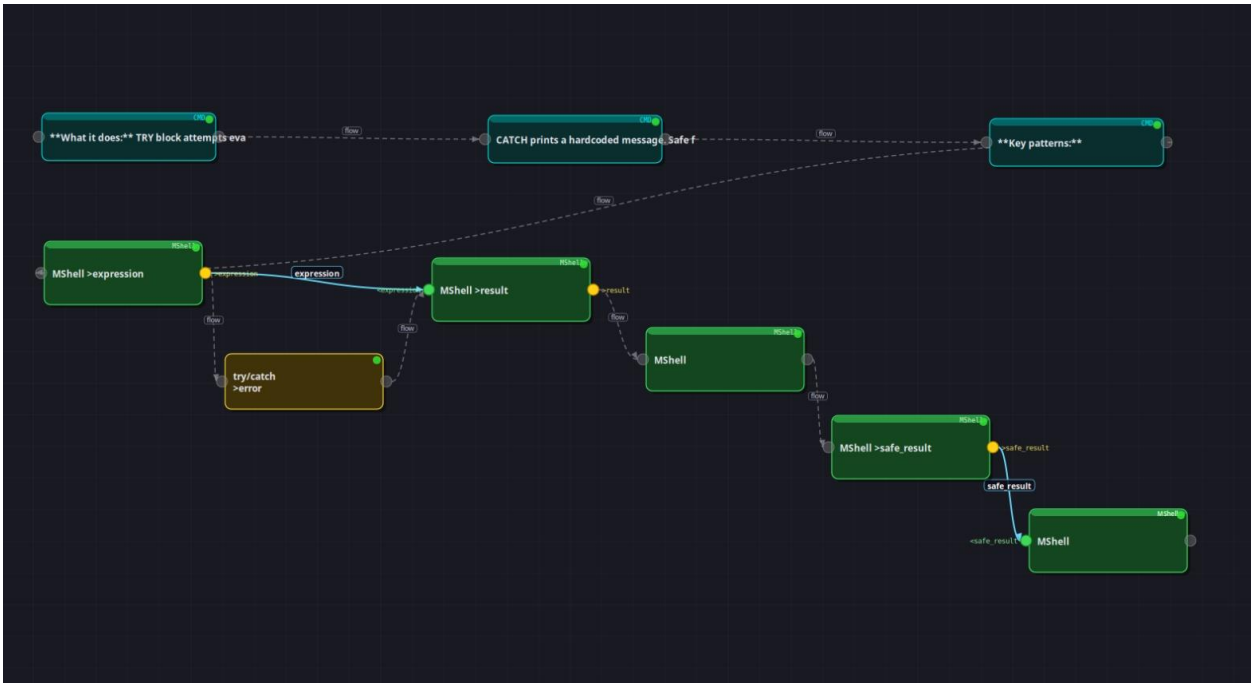
Code

```
mshell >expression print "sqrt(-1)"


mshell <expression >result set val = eval $expression print $val


mshell print "=== Caught error: try_block_failed ===" print "Switching to
fallback computation."


mshell >safe_result ollama1 "Compute the square root of 16. Reply with ONLY
the number."
mshell <safe_result print "=== Safe Result ===" print "sqrt(16) =
$safe_result"
```



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

What it does: Two-line stock dataset. SPLIT creates `dataset_1`/`dataset_2`. Two async LLMs analyze in parallel. Await synchronizes. @1 synthesizes a portfolio overview.

Key patterns: - SPLIT creates variables line by line; SPLIT and MERGE execute no code. - AWAIT before MERGE is mandatory.

Code

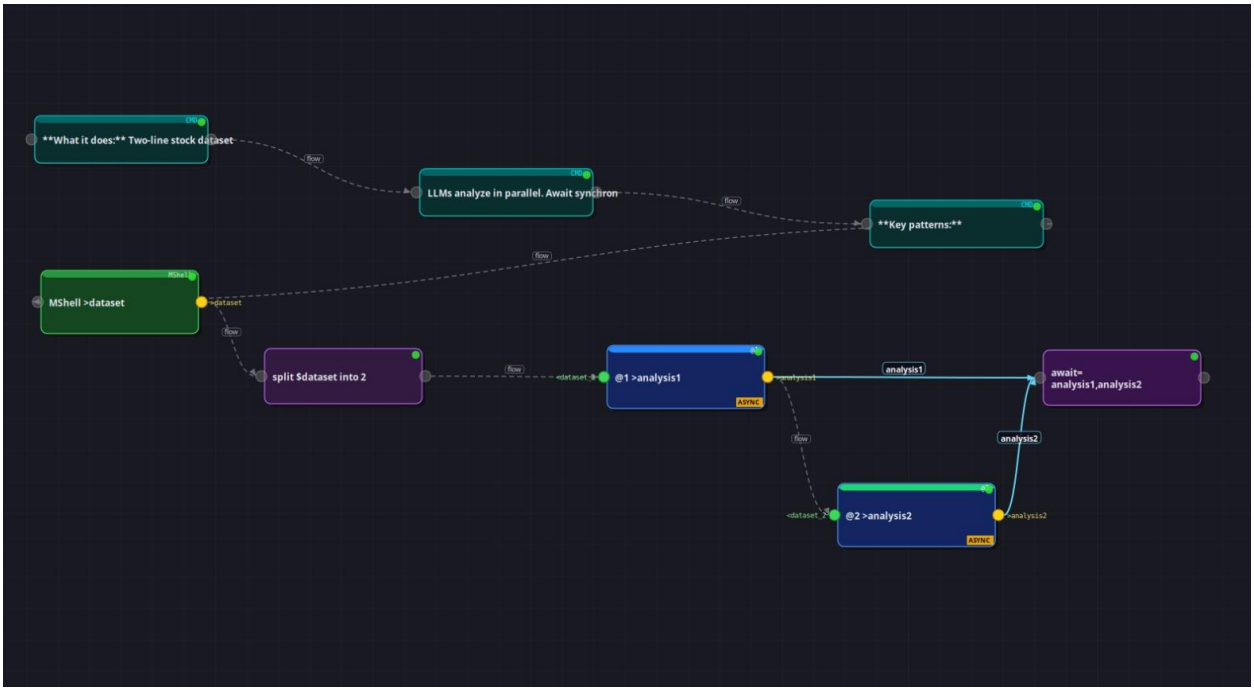
```
mshell >dataset print "AAPL:182,179,185,188,191" print
"MSFT:374,371,378,382,385"



mshell await=analysis1,analysis2



mshell <analysis1 <analysis2 <combined print "=== Stock 1 ===" print
$analysis1 print "" print "=== Stock 2 ===" print $analysis2 print "" print
"=== Portfolio Summary ===" print $combined
```



Pattern 17 CONFIG Node: Parameterized Pipeline

What it does: CONFIG documents parameters. mshell blocks set runtime values. @1 generates an explanation. @2 extracts keywords. mshell formats a report.

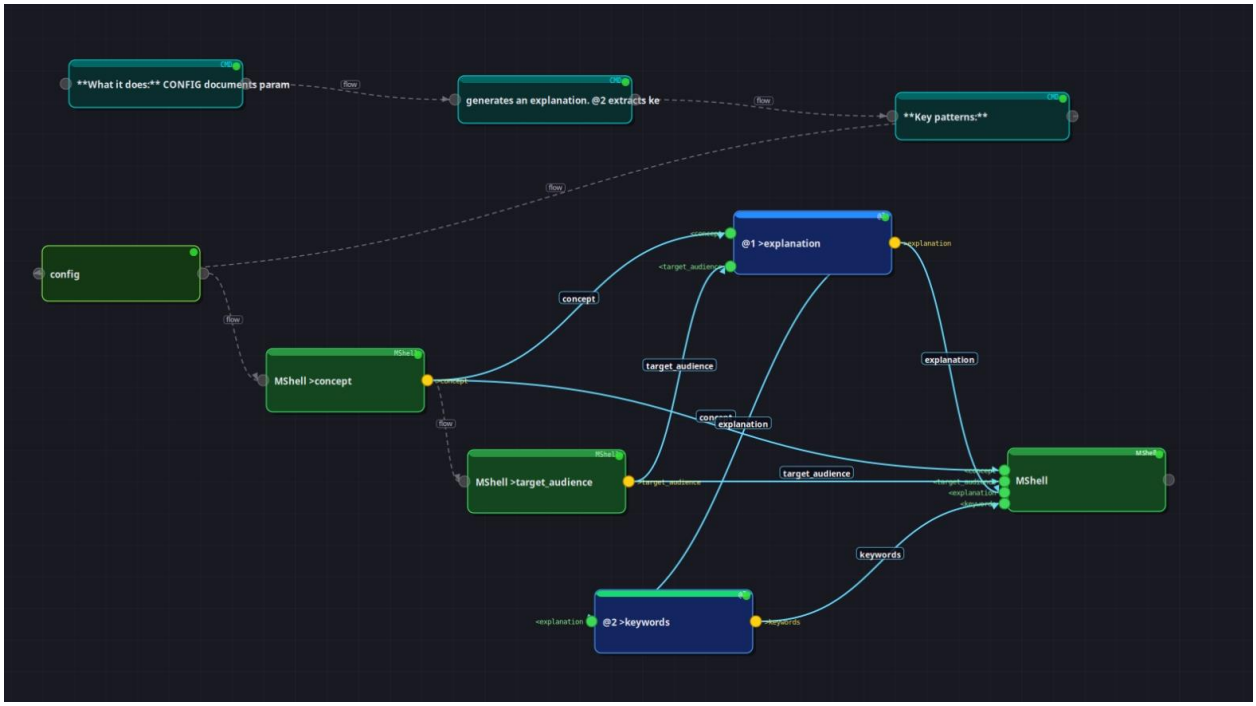
Key patterns: - CONFIG is documentation only 014 does NOT inject variables at runtime. - Always pair CONFIG entries with mshell blocks to set actual values.

Code

```
config concept=quantum entanglement target_audience=curious high school student max_words=60
mshell >concept print "quantum entanglement"
mshell >target_audience print "curious high school student"



mshell <concept <target_audience <explanation <keywords print "=== Workflow Report ===" print "Concept  : $concept" print "Audience : $target_audience" print "" print "Explanation:" print $explanation print "" print "Keywords : $keywords"
```



Pattern 18 — FOREACH + Async LLM: Parallel Batch Processing

Key patterns: - Use separate `print` per item — mshell does NOT interpret `\n` in strings. - Use `@1` for both async calls — @2 can be unreliable. - `await=` barrier lists all async outputs from the current iteration.

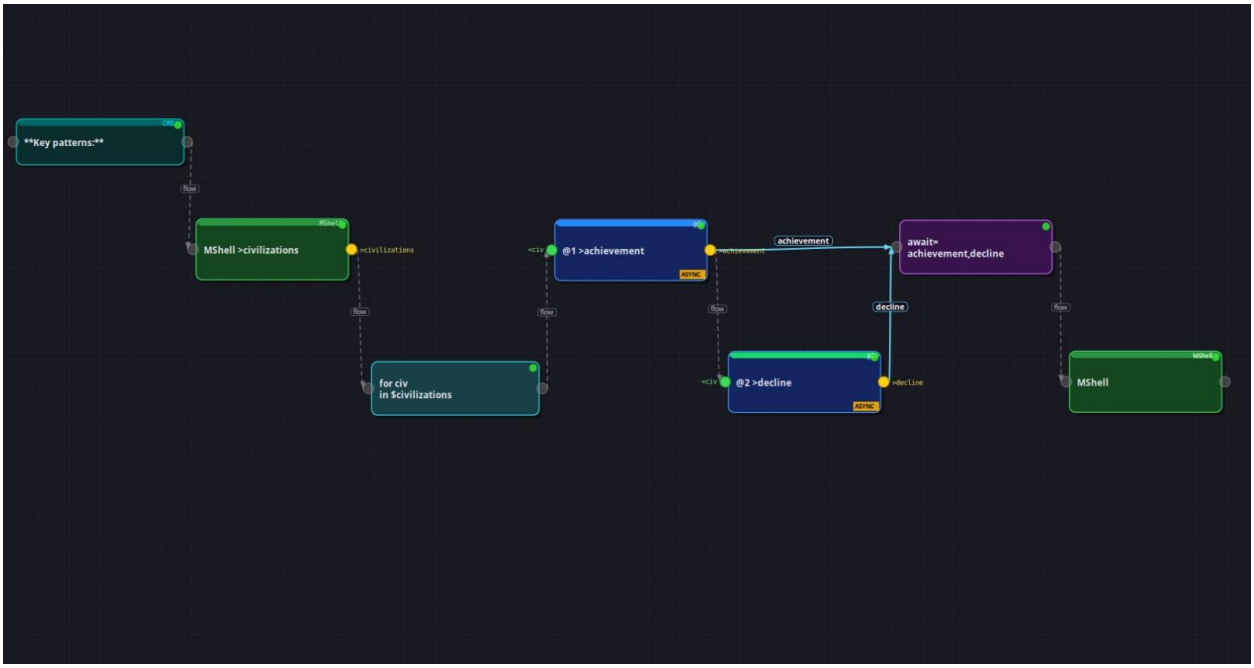
Code

```
mshell >civilizations print "Ancient Egypt" print "The Roman Empire" print
"The Maya"

mshell await=achievement,decline

mshell <achievement <decline print "Achievement : $achievement" print
"Decline     : $decline" print ""

print "=== Batch complete ==="
```



Pattern 19 WHILE Quality Gate: Generate Until Threshold

What it does: WHILE loop generates product taglines via @1 until @2 scores 265 8. Python block manages threshold check and status update.

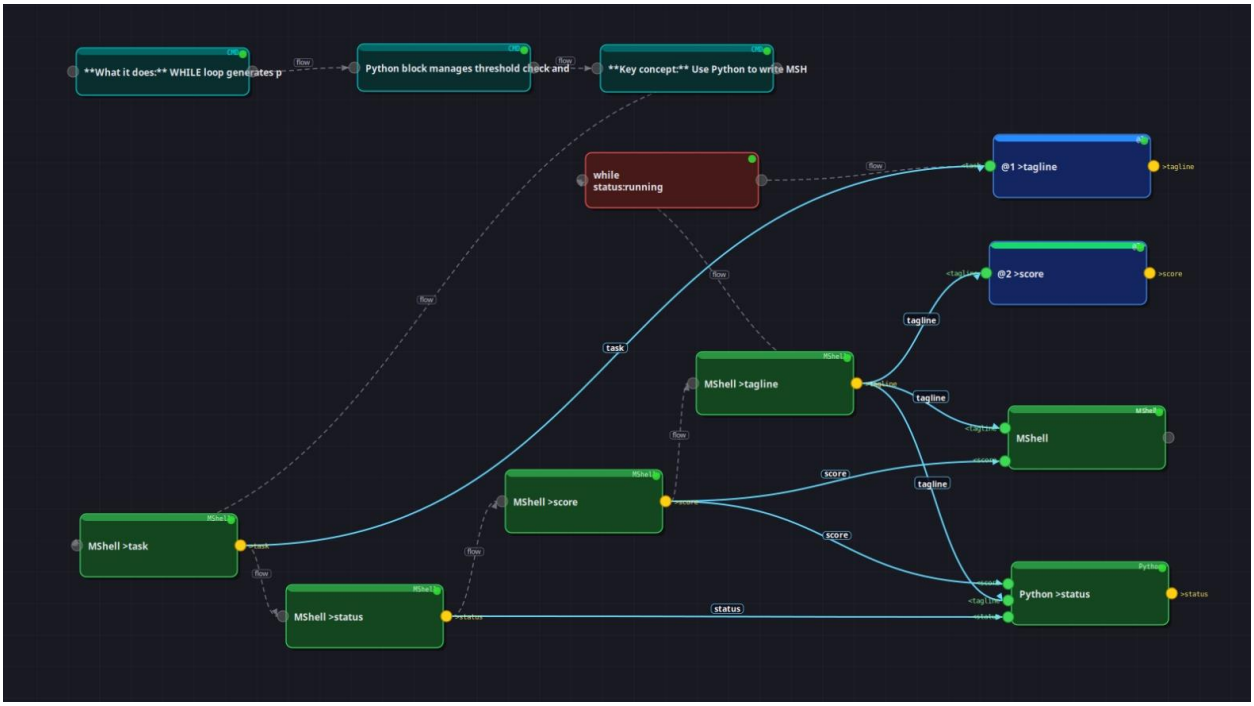
Key concept: Use Python to write MSH_VAR_status 014 mshell writefile unreliable in WHILE.

Code

```
mshell >task print "Write a punchy one-sentence tagline for a note-taking app called 'Clarity'."
mshell >status print "running"
mshell >score print "0"
mshell >tagline print ""
```

```python <score <tagline status import os score_path = os.environ['MSH_VAR_score'] status_path = os.environ['MSH_VAR_status'] tagline_path = os.environ['MSH_VAR_tagline']
score = open(score_path).read().strip() tagline = open(tagline_path).read().strip()
try: sc = int(''.join(c for c in score if c.isdigit())) except: sc = 0
print(f"Score={sc} | {tagline}")
if sc >= 8: open(status_path, 'w').write('done') print('done') else: open(status_path, 'w').write('running') print('running') ```
```

```
mshell <tagline <score print "=== Accepted tagline (score=$score) ===" print
$tagline
```



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

What it does: Computing history text split into 3 chunks. Three async @1 extract the main concept per chunk (MAP). Await synchronizes. @2 synthesizes (REDUCE).

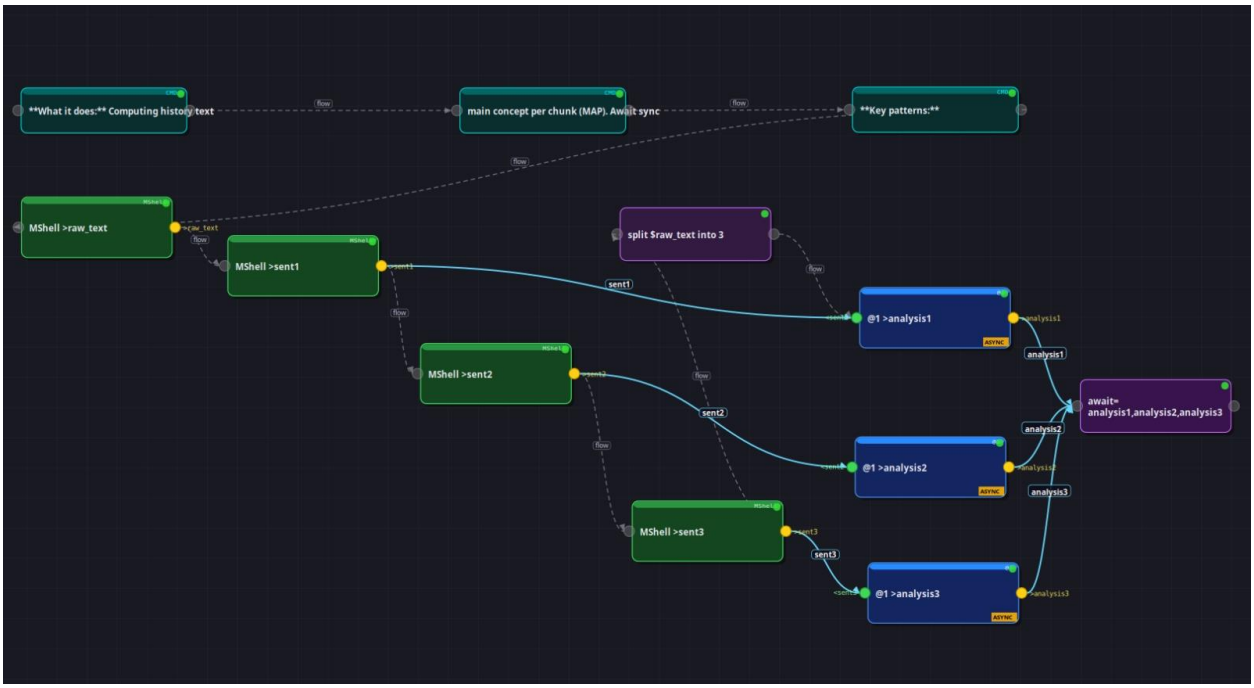
Key patterns: - Map: N parallel LLMs; total time = slowest chunk. - Reduce: one LLM receives all results via multiple `<invar` with `[varname]:` labels.

Code

```
mshell >raw_text print "Lua was designed at PUC-Rio in Brazil in 1993 as an
embeddable scripting language."
mshell >sent1 print "Lua was designed at PUC-Rio in Brazil in 1993 as an
embeddable scripting language."
mshell >sent2 print "It became one of the fastest scripting languages and is
embedded in millions of devices."
mshell >sent3 print "Its coroutine system enables cooperative multitasking
and metatables enable object-oriented programming."
```

```
mshell await=analysis1,analysis2,analysis3
```

```
mshell <analysis1 <analysis2 <analysis3 <summary print "=== Map ===" print
"Chunk 1: $analysis1" print "Chunk 2: $analysis2" print "Chunk 3: $analysis3"
print "" print "=== Reduce ===" print $summary
```



Pattern 21 TRY/CATCH + LOOP: Resilient Retry with Self-Correction

What it does: LOOP runs up to 3 times. @1 generates Python code. TRY executes it via Python subprocess. On success Python writes ok to result file and prints ok as last stdout line 014 LOOP checks last stdout line for exit condition.

Key concept: - LOOP checks **last stdout line** of last block in body. - Use Python (not mshell) for TRY block 014 reliable exec + result writing. - Do NOT put `>result` on TRY block 014 stdout capture breaks the line check.

Code

```
mshell >task print "Write Python code that prints the sum of numbers 1 to 10.
No imports needed. No explanation."
mshell >result print "fail"
```

```python <code <result import os, subprocess, tempfile
code_path = os.environ['MSH_VAR_code'] result_path = os.environ['MSH_VAR_result'] code
```

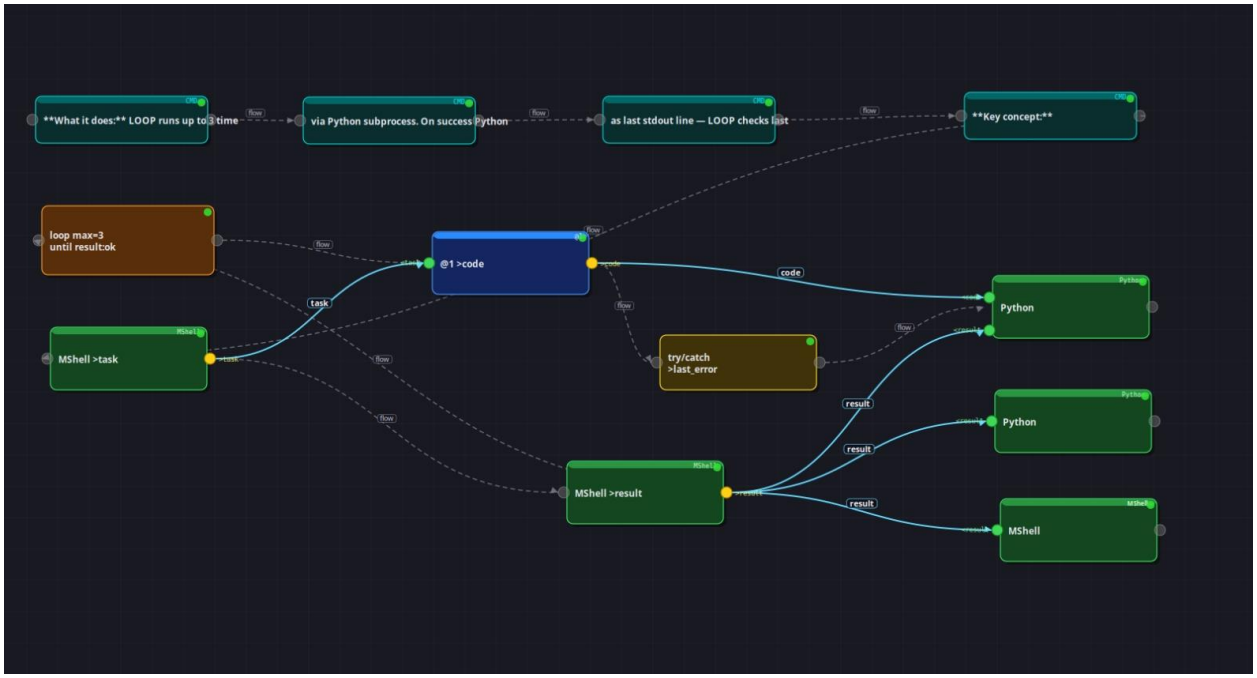
= open(code_path).read().strip()

with tempfile.NamedTemporaryFile(mode='w', suffix='.py', delete=False) as f: f.write(code) tmp = f.name

proc = subprocess.run(['python3', tmp], capture_output=True, text=True) if proc.returncode == 0: print(proc.stdout.strip()) open(result_path, 'w').write('ok') print('ok') else: print(proc.stderr.strip(), file=**import**('sys').stderr) open(result_path, 'w').write('fail') raise SystemExit(1) ```

```
python <result import os open(os.environ['MSH_VAR_result'],
'w').write('fail') print('fail')
```

```
mshell <result print "=== Final: $result ==="
```



Pattern 22 Multi-Variable Output: Structured Field Extraction

What it does: @1 responds to a scientific discovery description in strict 3-line format. A mshell block with 3 outputs parses the response and writes each field via `writefile`. @2 generates a "why it mattered" sentence.

Key patterns: - Multiple `>outvar` on CODE block: parser does NOT capture stdout 014 use `writefile`. - `MSH_VAR_*` env vars are pre-set by the parser before the block runs.

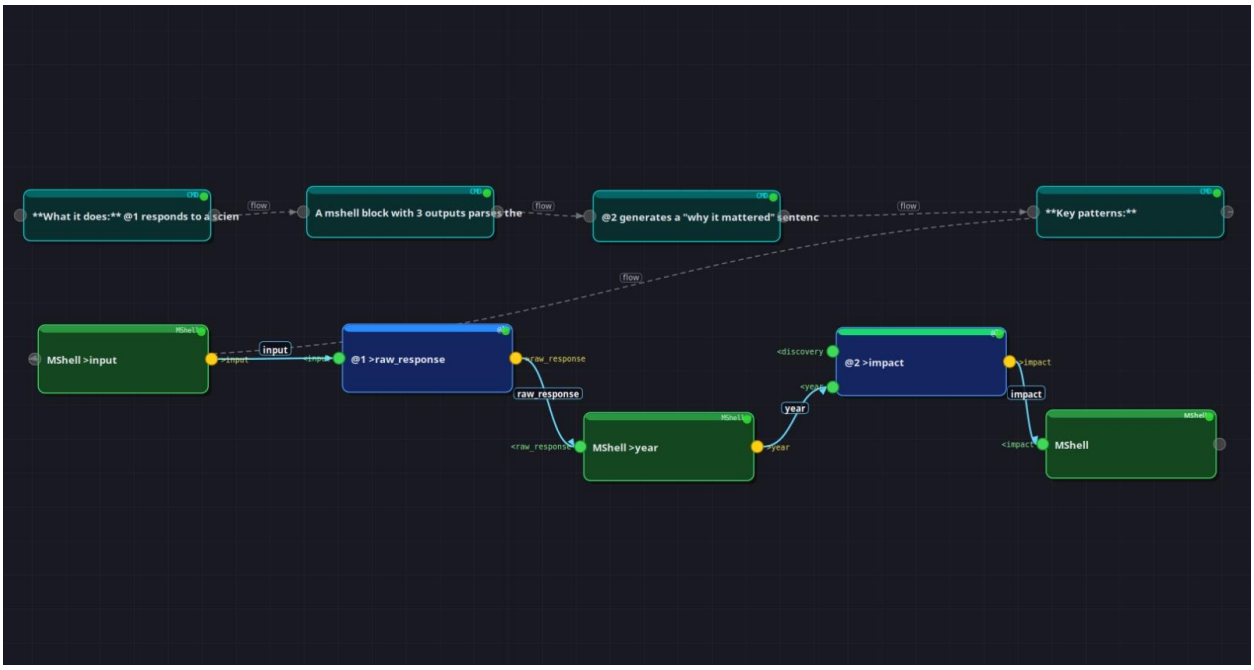
Code

```
mshell >input print "The discovery of penicillin by Alexander Fleming in 1928
revolutionized medicine by introducing the first true antibiotic, saving
```

```
hundreds of millions of lives."


mshell <raw_response >discovery >scientist >year set text = readfile
$MSH_VAR_raw_response set disc = "" set sci  = "" set yr   = "" set lines =
split($text, "\n") set i = 0 while eval $i < len($lines) set line =
$lines[$i] if startswith($line, "DISCOVERY:") set disc = trim(substr($line,
10)) end if startswith($line, "SCIENTIST:") set sci = trim(substr($line, 10))
end if startswith($line, "YEAR:") set yr = trim(substr($line, 5)) end set i =
eval $i + 1 end writefile $MSH_VAR_discovery "$disc" writefile
$MSH_VAR_scientist "$sci" writefile $MSH_VAR_year "$yr" print "Discovery :
$disc" print "Scientist : $sci" print "Year      : $yr"


mshell <impact print "" print "=== Why it mattered ===" print $impact
```



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

Key concept: Use Python to write MSH_VAR_status 014 mshell writefile unreliable in WHILE.

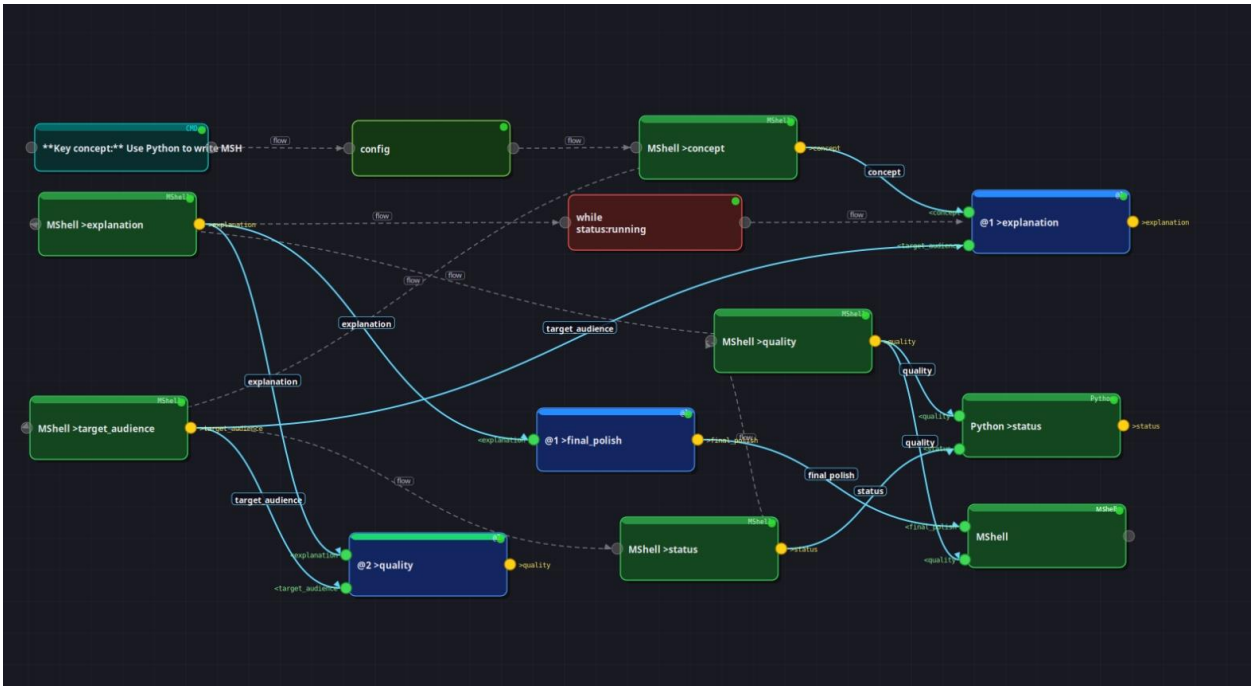
Code

```
config concept=Lua coroutines target_audience=high school student
quality_threshold=7
mshell >concept print "Lua coroutines"
mshell >target_audience print "high school student"
mshell >status print "running"
mshell >quality print "0"
```

```
mshell >explanation print ""
```

```
python <quality <status >status import os quality_path =
os.environ['MSH_VAR_quality'] status_path  = os.environ['MSH_VAR_status']
score = open(quality_path).read().strip() try: sc = int(''.join(c for c in
score if c.isdigit())) except: sc = 0 print(f"[Quality: {sc}]") if sc >= 7:
open(status_path, 'w').write('done') else: open(status_path,
'w').write('running')
```

```
mshell <final_polish <quality print "=== Final (score=$quality) ===" print
$final_polish
```



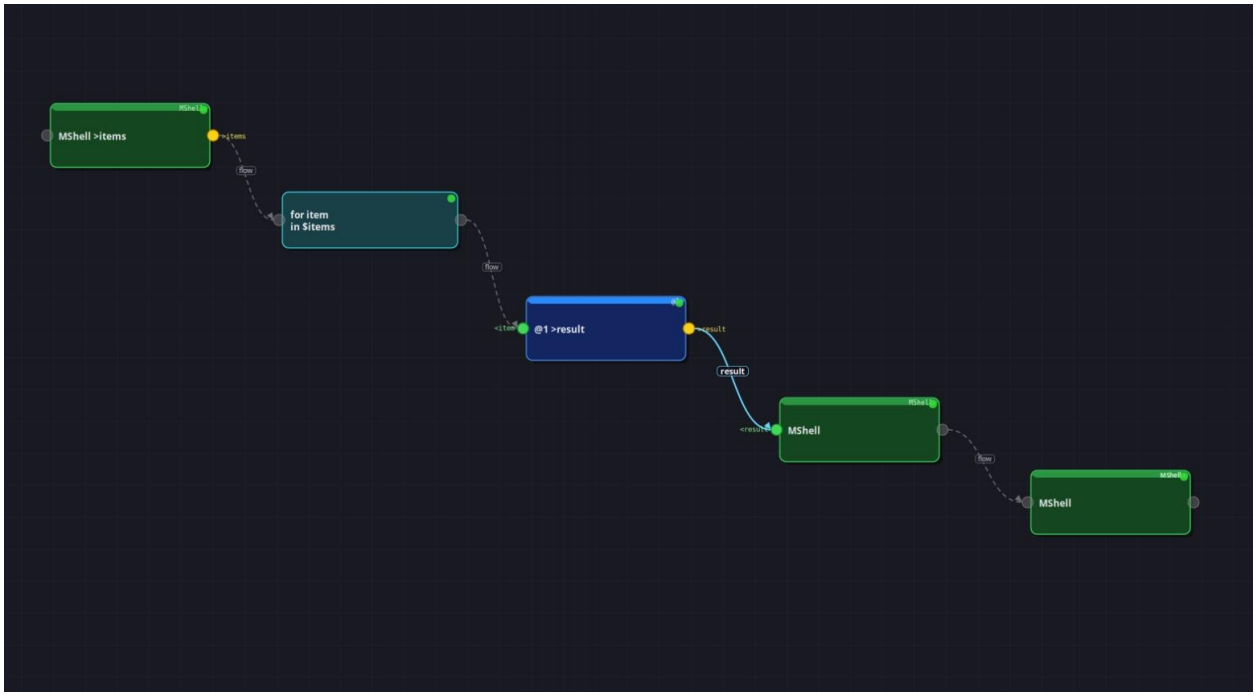
Pattern 24 FOREACH + TRY/CATCH: Fault-Tolerant Batch Processing

Code

```
mshell >items print "[Neutron Star]: density of a teaspoon equals mass of
Mount Everest" print "[broken item without closing bracket" print "[Black
Hole]: gravity so strong not even light can escape"
```

```
mshell <result print $result
```

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



Summary Table

#	Pattern	Nodes	Models	Key Technique
1	Linear Pipeline	—	—	ollama1 for math; $varname reads work
2	LLM in Middle	—	@1	LLM directive + mshell display
3	Fan-Out	—	@1	Same var, multiple consumers
4	Code Gen + Exec	—	@1	ollama1exec with >outvar
5	Two-LLM Review	—	@1, @2	Multi-<invar auto-labeled
6	Parallel 3 Models	—	@1–@3	Sequential LLM directives
7	Eval-Optimizer Loop	LOOP	@1, @2	Last stdout line = exit signal
8	Multi-Model Pipeline	—	@1, @2	ollama1exec for data; Python display
9	Routing	—	@1	if=route:VALUE conditional blocks
10	Full Pipeline	AWAIT	@1–@3	Async + await barrier
11	MShell Native AI	—	@1, @2	ollama1/2 inline in mshell blocks

#	Pattern	Nodes	Models	Key Technique
12	Async+Await+Synthesis	AWAIT	@1–@3	Parallel async + barrier
13	WHILE Counter	WHILE	@1	Python writes MSH_VAR_status
14	FOREACH List	FOREACH	@1	Separate print per item
15	TRY/CATCH	TRY/CATCH	—	eval triggers catch on invalid expr
16	SPLIT+MERGE	SPLIT, MERGE	@1, @2	Async parallel + await + synthesis
17	CONFIG Pipeline	CONFIG	@1, @2	CONFIG docs only; mshell sets values
18	FOREACH+Async	FOREACH, AWAIT	@1	Both async calls use @1
19	WHILE Quality Gate	WHILE	@1, @2	Python writes status on threshold
20	Map-Reduce	SPLIT, MERGE, AWAIT	@1, @2	3 async map + reduce synthesis
21	TRY/CATCH+LOOP	TRY/CATCH, LOOP	@1	Python executes generated code
22	Multi-Var Output	—	@1, @2	mshell writefile parses LLM response
23	CONFIG+WHILE+3M	CONFIG, WHILE	@1, @2	Python writes status; @1 polishes
24	FOREACH+TRY/CATCH	FOREACH	@1	LLM validates format — no TRY needed

---

Appendix I: Code examples for each pattern

/home/igor > Sent to mshell (1024 bytes)

Received from GUI editor:

Pattern 1 Linear Pipeline: Chain of MShell Transformations

What it does: A seed number flows through a pure mshell pipeline: square \u2192 prime check \u2192 format \u2192 report.

Key concept: Variables flow top-to-bottom. Use `$varname` directly in blocks that declare `<varname`. Use `ollama1` for computation that mshell cannot do natively.

Flow diagram

```
mshell >seed
 \u2193
mshell <seed >squared    (ollama1: compute square)
 \u2193
mshell <squared >verdict (ollama1: prime check)
 \u2193
mshell <seed <squared <verdict >report
 \u2193
mshell <report
```

Code

```mshell >seed
print 7
```

```mshell <seed >squared
ollama1 "Compute $seed * $seed. Reply with ONLY the integer result, nothing else."
```

```mshell <squared >verdict
ollama1 "Is $squared a prime number? Reply with exactly one word: PRIME or COMPOSITE."
```

```mshell <seed <squared <verdict >report

print "Number: $seed | Square: $squared | Classification: $verdict"

```

```mshell <report

print "=== Pipeline Result ==="

print $report

```

7

49COMPOSITENumber: 7 | Square: 49 | Classification: COMPOSITE

=== Pipeline Result ===

Number: 7 | Square: 49 | Classification: COMPOSITE

/home/igor >

/home/igor > Sent to mshell (887 bytes)

Received from GUI editor:

Pattern 2 LLM in the Middle: AI as a Transformation Layer

What it does: mshell generates CSV sensor data. LLM @1 analyzes for anomalies. mshell reads the report and prints a meta-summary.

Key concept: LLM directive injects variable contents into the prompt and stores the response in an output variable.

Flow diagram

```

mshell >data

 \u2193

@1 <data >analysis
```

\u2193

mshell <analysis

```

### Code

```mshell >data

print "sensor_id,timestamp,temp_c"

print "S01,08:00,22.1"

print "S01,08:05,22.4"

print "S01,08:10,41.7"

print "S01,08:15,22.0"

print "S01,08:20,22.3"

```

<!--@1 <data >analysis

The input is CSV sensor temperature data. Analyze it for anomalies.

Identify which reading is abnormal, estimate the spike magnitude,

and rate severity: LOW / MEDIUM / HIGH. Reply in 2-3 sentences.

-->

```mshell <analysis

print "=== Anomaly Report ==="

print $analysis

```

---

sensor_id,timestamp,temp_c

S01,08:00,22.1

S01,08:05,22.4

S01,08:10,41.7

S01,08:15,22.0
```

S01,08:20,22.3

The temperature reading of 41.7°C at 08:10 is clearly abnormal, as it spikes nearly 20 degrees above the baseline of around 22°C that all other readings maintain. This represents an approximate 90% increase from the normal temperature range, which could indicate a sensor malfunction, external heat source, or measurement error. I would rate this anomaly as HIGH severity due to the dramatic deviation from the established pattern.

=== Anomaly Report ===

The temperature reading of 41.7°C at 08:10 is clearly abnormal, as it spikes nearly 20 degrees above the baseline of around 22°C that all other readings maintain. This represents an approximate 90% increase from the normal temperature range, which could indicate a sensor malfunction, external heat source, or measurement error. I would rate this anomaly as HIGH severity due to the dramatic deviation from the established pattern.

/home/igor >

/home/igor > Sent to mshell (936 bytes)

Received from GUI editor:

Pattern 3 Fan-Out: One Source, Many Consumers

What it does: A single variable is consumed by two mshell blocks and one LLM

independently. Each reads the same unchanged file.

Key concept: Multiple blocks can reference `<varname`. All consumers read the same file.

Flow diagram

```

mshell >quote

 \u2193       \u2193          \u2193

mshell     mshell       @1 <quote

<quote     <quote       >sentiment

```

Code

```mshell >quote
```

print "The unexamined life is not worth living."

```

```mshell <quote

print "Word count: $(len $(split($quote, ' ')))"

print "Quote: $quote"

```

```mshell <quote

ollama1 "Reverse the word order of this sentence: $quote. Reply with ONLY the reversed sentence."

```

<!--@1 <quote >sentiment

The input is a philosophical quote.

In one sentence: identify its philosophical era and emotional tone.

-->

```mshell <sentiment

print "=== Sentiment Analysis ==="

print $sentiment

```

The unexamined life is not worth living.

Word count: $(len $(split(The unexamined life is not worth living., ' ')))

Quote: The unexamined life is not worth living.

living. worth not is life unexamined The

This quote from ancient Greek philosophy (attributed to Socrates) carries a serious, challenging tone that urges deep self-reflection and questions the value of an unreflective existence.

=== Sentiment Analysis ===
```

This quote from ancient Greek philosophy (attributed to Socrates) carries a serious, challenging tone that urges deep self-reflection and questions the value of an unreflective existence.

/home/igor >

Sent to mshell (905 bytes)

Received from GUI editor:

Pattern 4 LLM Code Generation - Execute via Variable

What it does: `ollama1exec` asks LLM @1 to generate and execute mshell code.

Result is captured into `>result` and displayed via `readfile`.

Key concept: Always use `>outvar` with `ollama1exec` to prevent mshell from crashing.

Read the result back with `readfile $MSH_VAR_result` \u2014 not `$result`.

Flow diagram

```

mshell >task

 \u2193

mshell <task >result   (ollama1exec: generate + execute)

 \u2193

mshell <result        (readfile to display)

```

Code

```mshell >task

print "Generate the Collatz sequence starting from 27. Print each number on its own line until you reach 1."

```

```mshell <task >result

ollama1exec "Generate and execute mshell code for this task: $task. Use mshell syntax: set, while, if, eval, print."
```

```

```mshell <result

set text = readfile $MSH_VAR_result

print "=== Result ==="

print $text

```

Generate the Collatz sequence starting from 27. Print each number on its own line until you reach 1.

I'll generate mshell code to create the Collatz sequence starting from 27.

```mshell

set n 27

while $n -ne 1

  print $n

  if $n % 2 -eq 0

    eval n = $n / 2

  else

    eval n = 3 * $n + 1

  end

end

print $n

```

Let me execute this step by step:

```mshell

set n 27

while $n -ne 1

  print $n

```
    if $n % 2 -eq 0
        eval n = $n / 2
    else
        eval n = 3 * $n + 1
    end
end
print $n
```

Output:

```
27
82
41
124
62
31
94
47
142
71
214
107
322
161
484
242
121
```

364

182

91

274

137

412

206

103

310

155

466

233

700

350

175

526

263

790

395

1186

593

1780

890

445

1336

668

334

167

502

251

754

377

1132

566

283

850

425

1276

638

319

958

479

1438

719

2158

1079

3238

1619

4858

2429

7288

3644

1822

911

2734

1367

4102

2051

6154

3077

9232

4616

2308

1154

577

1732

866

433

1300

650

325

976

488

244

122

61

184

92

46

23

70

35

106

53

160

80

40

20

10

5

16

8

4

2

1

```

The Collatz sequence starting from 27 takes 111 steps to reach 1, with the highest number in the sequence being 9232.=== Result ===

/home/igor >

---

/home/igor > Sent to mshell (1160 bytes)

Received from GUI editor:

---

## Pattern 5 Two-LLM Review Chain: Generate - Review - Improve

**What it does:** @1 generates a riddle. @2 critiques it. @1 receives both and produces

an improved version.

**Key concept:** Multiple `<invar` auto-labeled as `[varname]:` in the prompt.

### Flow diagram
```

```

mshell >task

 \u2193

@1 <task >riddle

 \u2193

@2 <riddle >review

 \u2193

@1 <riddle <review >improved

 \u2193

mshell <improved

```

Code

```mshell >task

print "Write a short riddle about time. Include the answer at the end after 'Answer:'. Make it 3-4 lines."

```

<!--@1 <task >riddle

The input is a creative writing task. Complete it exactly as described.

Reply with only the riddle and its answer. No preamble.

-->

<!--@2 <riddle >review

The input is a riddle. Review it critically in one paragraph:

Is the answer guessable? Is the wordplay clever? What specific fix would improve it?

-->

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

The first input is your original riddle. The second is a critic's feedback.

Rewrite the riddle incorporating the critique. 3-4 lines with Answer. Return only the riddle.

-->

```mshell <improved

print "=== Final Improved Riddle ==="

print $improved

```

Write a short riddle about time. Include the answer at the end after 'Answer:'. Make it 3-4 lines.

I have no body, yet I fly,

I cannot be caught, though many try.

I heal all wounds, but bring decay,

What am I that rules each day?

Answer: Time

The answer is very guessable: the clues (no body yet flies, can\u2019t be caught, heals all wounds yet causes decay, rules each day) are all classic \u201ctime\u201d motifs, so most readers will jump to it quickly. The wordplay is straightforward rather than clever\u2014there\u2019s no twist, misdirection, or unusual image, just standard poetic descriptions of time. To improve it, add one unexpected or ambiguous clue that could also fit another answer, then subtly steer it back toward \u201ctime\u201d; for example, change \u201cI cannot be caught, though many try\u201d to something more image-rich like \u201cI slip through fingers, net, and snare,\u201d which keeps the meaning but adds a more vivid, less cliché expression.

I have no body, yet I fly,

I slip through fingers, net, and snare.

I mend the heart but steal your hair,

What am I that's always there?

Answer: Time

=== Final Improved Riddle ===

I have no body, yet I fly,

I slip through fingers, net, and snare.

I mend the heart but steal your hair,

What am I that's always there?

Answer: Time

/home/igor >

/home/igor > Sent to mshell (1042 bytes)

Received from GUI editor:

Pattern 6 Parallel 3-Model Query: Three Philosophical Traditions

What it does: The same question sent to all three models. Each adopts a distinct philosophical tradition. mshell collects and displays all three.

Key concept: Three LLM blocks with different model numbers all read `<question` and write to distinct output variables.

Flow diagram

```
mshell >question
 \u2193
@1 <question >ans1   @2 <question >ans2   @3 <question >ans3
 \u2193
mshell <ans1 <ans2 <ans3
```

Code

```mshell >question
print "What should a person do when faced with an outcome they cannot control?"
```

```
<!--@1 <question >ans1
You are a Stoic philosopher. Answer in exactly one sentence from the Stoic perspective.
-->
```

<!--@2 <question >ans2

You are a Zen master. Answer in exactly one sentence using Zen philosophy.

-->

<!--@3 <question >ans3

You are a modern existentialist. Answer in exactly one sentence.

-->

```mshell <ans1 <ans2 <ans3

print "[Stoicism]      $ans1"

print "[Zen]          $ans2"

print "[Existentialism] $ans3"

```

What should a person do when faced with an outcome they cannot control?

Focus on what is within your power\u2014your thoughts, judgments, and responses\u2014while accepting what lies beyond your control with equanimity, for this is the path to inner peace and wisdom.

Sit quietly with the arising and passing of the outcome, allowing it to be exactly as it is while you attend only to the next clear, compassionate step before you.

Embrace your radical freedom to create meaning through chosen action, even when the outcome remains beyond your control.

[Stoicism]   Focus on what is within your power\u2014your thoughts, judgments, and responses\u2014while accepting what lies beyond your control with equanimity, for this is the path to inner peace and wisdom.

[Zen]   Sit quietly with the arising and passing of the outcome, allowing it to be exactly as it is while you attend only to the next clear, compassionate step before you.

[Existentialism] Embrace your radical freedom to create meaning through chosen action, even when the outcome remains beyond your control.

/home/igor >

/home/igor > Sent to mshell (1423 bytes)

Received from GUI editor:

Pattern 7 Evaluator-Optimizer Loop: Generate Until Accepted

What it does: @1 generates a mnemonic for the 8 planets. @2 evaluates strictly:

`ACCEPTED` or `REJECTED: reason`. Loop repeats max 3 times until accepted.

Key concept: `<!--@loop max=N until=verdict:ACCEPTED-->` exits early on acceptance.

Flow diagram

```

mshell >task

 \u2193

[LOOP max=3 until=verdict:ACCEPTED]

 @1 <task <verdict >mnemonic

 mshell <mnemonic

 @2 <mnemonic >verdict

 mshell <verdict

[END_LOOP]

 \u2193

mshell <mnemonic
```

Code

```mshell >task

print "Create a mnemonic sentence where each word starts with the first letter of a planet in order: Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune."

```

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

<!--@1 <task <verdict >mnemonic

First input: the mnemonic task. Second input: previous verdict (or empty on first attempt).

If REJECTED, fix the specific issue mentioned. Return ONLY the mnemonic sentence.

-->

```mshell <mnemonic

print "--- Attempt: $mnemonic"

```

<!--@2 <mnemonic >verdict

Check strictly: does each word start with M, V, E, M, J, S, U, N in that exact order?

If YES and the sentence is grammatically coherent: reply with exactly one word: ACCEPTED

If NO: reply REJECTED: <one sentence explaining what is wrong>

-->

```mshell <verdict

print "--- Verdict: $verdict"

```

<!--@end_loop-->

```mshell <mnemonic

print "=== Accepted: $mnemonic ==="

```

[loop] Starting loop: max=1 until=verdict:ACCEPTED

Create a mnemonic sentence where each word starts with the first letter of a planet in order: Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune.

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

My very educated mother just served us nachos.

--- Attempt: My very educated mother just served us nachos.

REJECTED: The sixth word starts with "s" instead of the required "J".

--- Verdict: REJECTED: The sixth word starts with "s" instead of the required "J".

[loop] until check: last_line='REJECTED: The sixth word starts with "s" instead of the required "J".' expected='ACCEPTED'

[loop] Iteration 1/3 \u2014 condition not met, looping back

My very educated mother just jumped under nets.

--- Attempt: My very educated mother just jumped under nets.

REJECTED: The sixth word starts with "u" instead of the required "S".

--- Verdict: REJECTED: The sixth word starts with "u" instead of the required "S".

[loop] until check: last_line='REJECTED: The sixth word starts with "u" instead of the required "S".' expected='ACCEPTED'

[loop] Iteration 2/3 \u2014 condition not met, looping back

My very educated mother just saw unusual newts.

--- Attempt: My very educated mother just saw unusual newts.

ACCEPTED

--- Verdict: ACCEPTED

[loop] Exiting loop after 3 iteration(s). reason: until condition met

=== Accepted: My very educated mother just saw unusual newts. ===

/home/igor >

/home/igor > Sent to mshell (889 bytes)

Received from GUI editor:

Pattern 8 Multi-Model Pipeline: Compute - Analyze - Summarize

Code

```mshell >raw

ollama1exec "Print the first 10 Fibonacci numbers separated by spaces on one line. Print ONLY the numbers, nothing else: 1 1 2 3 5 8 13 21 34 55"

```

```mshell >analysis

ollama1exec "In one sentence, describe what the first 10 Fibonacci numbers (1 1 2 3 5 8 13 21 34 55) reveal about exponential growth and the golden ratio."
```

```

```mshell >headline

ollama1exec "Give a 5-word tweet-style headline about Fibonacci numbers and the golden ratio. Nothing else."

```

```python <raw <analysis <headline

import os

raw      = open(os.environ['MSH_VAR_raw']).read().strip()

analysis = open(os.environ['MSH_VAR_analysis']).read().strip()

headline = open(os.environ['MSH_VAR_headline']).read().strip()

print(f"[Data]     {raw}")

print(f"[Analysis] {analysis}")

print(f"[Headline] {headline}")

```

---

1 1 2 3 5 8 13 21 34 55The first 10 Fibonacci numbers reveal that while the sequence grows exponentially at a rate approaching the golden ratio (\u03c6 \u2248 1.618), with consecutive terms' ratios converging toward this value (55/34 \u2248 1.618), demonstrating how this mathematical constant emerges naturally from the simple rule of adding consecutive terms.Fibonacci spirals reveal golden ratio magic/home/igor > [Data] 1 1 2 3 5 8 13 21 34 55

[Analysis] The first 10 Fibonacci numbers reveal that while the sequence grows exponentially at a rate approaching the golden ratio (\u03c6 \u2248 1.618), with consecutive terms' ratios converging toward this value (55/34 \u2248 1.618), demonstrating how this mathematical constant emerges naturally from the simple rule of adding consecutive terms.

[Headline] Fibonacci spirals reveal golden ratio magic

/home/igor >

/home/igor > Sent to mshell (1167 bytes)

Received from GUI editor:
```

Pattern 9 Routing: LLM Classifies - Conditional Branch Executes

What it does: @1 classifies a query into SCIENCE / HISTORY / PHILOSOPHY. Only the matching branch executes. All three test cases shown.

Key concept: `if=varname:expected_value` makes any block conditional.

Flow diagram

```
mshell >input
 \u2193
@1 <input >route
 \u2193
@1 <input if=route:SCIENCE >answer
@1 <input if=route:HISTORY >answer
@1 <input if=route:PHILOSOPHY >answer
 \u2193
mshell <route <answer
```

Code \u2014 Test 1: SCIENCE

```mshell >input
print "Why does water expand when it freezes?"
```

<!--@1 <input >route
Classify this question into exactly one word: SCIENCE, HISTORY, or PHILOSOPHY.
Reply with ONLY that one word.
-->

<!--@1 <input if=route:SCIENCE >answer
Answer this science question in two sentences. Be precise and factual.

-->

<!--@1 <input if=route:HISTORY >answer

Answer this history question in two sentences with key dates and context.

-->

<!--@1 <input if=route:PHILOSOPHY >answer

Answer this philosophical question in two sentences with a key thinker reference.

-->

```mshell <route <answer

print "Route: $route"

print "Answer: $answer"

```

Why does water expand when it freezes?

SCIENCE

When water freezes, its molecules form a rigid hexagonal crystalline structure that is less dense than liquid water, where molecules are more randomly arranged and can pack more closely together. This organized ice crystal lattice takes up approximately 9% more volume than the same mass of liquid water, causing expansion.

Route: SCIENCE

Answer: When water freezes, its molecules form a rigid hexagonal crystalline structure that is less dense than liquid water, where molecules are more randomly arranged and can pack more closely together. This organized ice crystal lattice takes up approximately 9% more volume than the same mass of liquid water, causing expansion.

/home/igor >

/home/igor > Sent to mshell (1674 bytes)

Received from GUI editor:

Pattern 10 Full Pipeline: All Patterns Combined

What it does: mshell computes perfect squares. Two LLMs run async (analyst + poet). Await barrier synchronizes. Third LLM synthesizes. Final mshell formats the output.

Key concept: Patterns compose naturally \u2014 every block reads named variables.

Flow diagram

```
mshell >raw_data       (perfect squares)
  \u2193
mshell <raw_data >stats
  \u2193
@1 <stats >analysis async \u2500\u2510
@2 <raw_data >poem async   \u2500\u2524
mshell await=analysis,poem \u2500\u2518
  \u2193
@1 <analysis <poem >combined
  \u2193
mshell <combined
```

Code

```mshell >raw_data
set out = ""
set i = 1
while eval $i <= 8
  set sq = eval $i * $i
  if eval $i == 1
    set out = $sq
  else
    set out = "$out,$sq"
```

~~~~
  end

  set i = eval $i + 1

end

print $out

```

```mshell <raw_data >stats

ollama1 "The input is a CSV of perfect squares: $raw_data. Compute min, max, and mean. Reply in one line: min=X max=Y mean=Z"

```

<!--@1 <stats >analysis async

The input contains statistics about perfect squares.

In one sentence, describe what is mathematically interesting about this sequence.

-->

<!--@2 <raw_data >poem async

Write a 2-line rhyming poem about perfect squares using these numbers: $raw_data

-->

```mshell await=analysis,poem

```

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

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

-->

```mshell <combined

print "\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550"

print $combined
~~~~

print "\u2550"

```

----------------------------------------

[async llm] Launched PID 381739 \u2192 var=analysis (model @1)

[async llm] Launched PID 381740 \u2192 var=poem (model @2)

[async] await= barrier: waiting for vars: analysis,poem

[async] Waiting for PID 381739 (var=analysis)...

[async] PID 381739 done (var=analysis)

Perfect squares (1, 4, 9, 16, 25, ...) are mathematically interesting because the differences between consecutive perfect squares form the sequence of odd numbers (3, 5, 7, 9, ...), revealing the fundamental relationship that the nth perfect square equals the sum of the first n odd numbers.

Perfect squares (1, 4, 9, 16, 25, 36, 49, 64, 81, 100) create an elegant mathematical symphony where each note represents the sum of consecutive odd numbers, revealing how these "perfect patterns that fill the air" emerge from the fundamental principle that the nth square equals the sum of the first n odd integers.

\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550

Perfect squares (1, 4, 9, 16, 25, 36, 49, 64, 81, 100) create an elegant mathematical symphony where each note represents the sum of consecutive odd numbers, revealing how these "perfect patterns that fill the air" emerge from the fundamental principle that the nth square equals the sum of the first n odd integers.

\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\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 >

--------------------------------------------------------------------------------

/home/igor > Sent to mshell (1056 bytes)
```

Received from GUI editor:

Pattern 11 MShell Native AI Node: ollama Commands in mshell Blocks

What it does: Uses `ollama1`/`ollama2` commands inside mshell blocks. Topic and style are set via mshell. First block calls ollama1 inline. Second passes result to ollama2.

Key concept: `ollama1 "prompt $varname"` calls model @1 inline; `$varname` from `<invar` is available directly in the prompt string.

Flow diagram

```
mshell >topic
mshell >style
 \u2193
mshell <topic <style >explanation   (ollama1 inline)
 \u2193
mshell <explanation >keywords      (ollama2 inline)
 \u2193
mshell <explanation <keywords
```

Code

```mshell >topic
print "quantum superposition"
```

```mshell >style
print "vivid analogy for a 12-year-old"
```

```mshell <topic <style >explanation
ollama1 "Explain $topic using $style. One sentence only."
```

```

```mshell <explanation >keywords

ollama2 "Extract exactly 3 keywords from: $explanation. Reply with 3 words comma-separated."

```

```mshell <explanation <keywords

print "Explanation: $explanation"

print "Keywords   : $keywords"

```

---

quantum superposition

vivid analogy for a 12-year-old

Imagine a coin spinning in the air - while it's spinning, it's neither heads nor tails but somehow both at the same time, and quantum particles exist in this "spinning" state until we measure them and force them to "land" on one specific answer.coin, quantum, measureExplanation: Imagine a coin spinning in the air - while it's spinning, it's neither heads nor tails but somehow both at the same time, and quantum particles exist in this "spinning" state until we measure them and force them to "land" on one specific answer.

Keywords   : coin, quantum, measure

/home/igor >

/home/igor > Sent to mshell (1603 bytes)

Received from GUI editor:

Pattern 12 Async Parallel 3 Models + Await Barrier + Synthesis

What it does: Three LLM models answer a question asynchronously (neuroscience / computation / phenomenology). Await barrier synchronizes. Fourth LLM synthesizes.

Key concept: `async` launches in background. `mshell await=v1,v2,v3` blocks until all variables are written. Total time = slowest model, not the sum.
```

Flow diagram

```
mshell >question
 \u2193
@1 <question >ans1 async \u2500\u2510
@2 <question >ans2 async \u2500\u2524
@3 <question >ans3 async \u2500\u2518
mshell await=ans1,ans2,ans3
 \u2193
@1 <ans1 <ans2 <ans3 >final
 \u2193
mshell <final
```

Code

```mshell >question
print "What is consciousness and can a machine ever truly have it?"
```

<!--@1 <question >ans1 async
Answer from a neuroscience perspective in exactly one sentence.
You MUST respond \u2014 do not leave the response empty.
-->

<!--@2 <question >ans2 async
Answer from a computational theory perspective in exactly one sentence.
You MUST respond \u2014 do not leave the response empty.
-->

<!--@3 <question >ans3 async
Answer from a phenomenological philosophy perspective in exactly one sentence.

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

-->

```mshell await=ans1,ans2,ans3
```

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

Three perspectives on consciousness are provided. Synthesize them into one

perfect sentence that bridges neuroscience, computation, and phenomenology.

-->

```mshell <ans1 <ans2 <ans3 <final
print "[Neuroscience]  $ans1"
print "[Computation]   $ans2"
print "[Phenomenology] $ans3"
print ""
print "=== Synthesis ==="
print $final
```

What is consciousness and can a machine ever truly have it?

[async llm] Launched PID 381798 \u2192 var=ans1 (model @1)

[async llm] Launched PID 381799 \u2192 var=ans2 (model @2)

[async llm] Launched PID 381801 \u2192 var=ans3 (model @3)

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

[async] Waiting for PID 381798 (var=ans1)...

[async] PID 381798 done (var=ans1)

From a neuroscience perspective, consciousness appears to emerge from complex neural network interactions and information integration in the brain, suggesting that machines could theoretically achieve consciousness if they can replicate the relevant computational processes and neural architectures that generate subjective experience.

Consciousness emerges when complex neural networks integrate self-referential information to create the subjective experience of radical freedom\u2014a phenomenon that machines might achieve by replicating the computational architectures that transform objective processes into the felt reality of choosing meaning through action.

[Neuroscience] From a neuroscience perspective, consciousness appears to emerge from complex neural network interactions and information integration in the brain, suggesting that machines could theoretically achieve consciousness if they can replicate the relevant computational processes and neural architectures that generate subjective experience.

[Computation] From a computational theory perspective, consciousness can be modeled as the integrated, self-referential processing of information within a system, and in principle a machine implementing the right computational architecture and dynamics could exhibit all the functional properties we associate with being truly conscious.

[Phenomenology] Embrace your radical freedom to create meaning through chosen action, even when the outcome remains beyond your control.

=== Synthesis ===

Consciousness emerges when complex neural networks integrate self-referential information to create the subjective experience of radical freedom\u2014a phenomenon that machines might achieve by replicating the computational architectures that transform objective processes into the felt reality of choosing meaning through action.

/home/igor >

Sent to mshell (1201 bytes)

Received from GUI editor:

Pattern 13 Iterative Counter with LLM Commentary

What it does: counter=0, status=running. WHILE loop. Python block increments

counter and writes status via MSH_VAR_* files directly. @1 gives math fact per number.

Key concept: Use Python for counter increment in mshell WHILE loops \u2014

Python can write to MSH_VAR_* files reliably where mshell writefile fails.

Code

```mshell >status

print "running"
```

```

```mshell >counter
print "0"
```

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

```python <counter <status >counter >status
import os
counter_path = os.environ['MSH_VAR_counter']
status_path  = os.environ['MSH_VAR_status']
val    = int(open(counter_path).read().strip() or '0')
newval = val + 1
open(counter_path, 'w').write(str(newval))
open(status_path,  'w').write('done' if newval >= 4 else 'running')
print(newval)
```

<!--@1 <counter >comment
The input contains a single integer.
In one sentence, share a fascinating mathematical property of that specific number.
You MUST respond \u2014 do not leave the response empty.
-->

```mshell <comment
print "[iter $counter] $comment"
```

<!--@end_while-->

```mshell <counter
print "=== WHILE complete. Final counter = $counter ==="
```
```

---

running

0

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

1

1 is the only positive integer that is equal to its own factorial, since 1! = 1.

[iter] 1 is the only positive integer that is equal to its own factorial, since 1! = 1.

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

2

2 is the only even prime number, making it unique among all prime numbers since every other even number greater than 2 is divisible by 2 and therefore composite.

[iter] 2 is the only even prime number, making it unique among all prime numbers since every other even number greater than 2 is divisible by 2 and therefore composite.

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

3

3 is the smallest odd prime number and the only prime number that is one less than a perfect square (since $3 = 4 - 1 = 2^2 - 1$).

[iter] 3 is the smallest odd prime number and the only prime number that is one less than a perfect square (since $3 = 4 - 1 = 2^2 - 1$).

[while] iteration 4 \u2014 condition met, executing body

4

4 is the smallest composite number and the only number that equals both the sum and product of its proper divisors ($1 + 1 = 2$ and $1 \times 1 = 1$, but more notably, $4 = 2^2$ and is the first perfect square greater than 1).

[iter] 4 is the smallest composite number and the only number that equals both the sum and product of its proper divisors ($1 + 1 = 2$ and $1 \times 1 = 1$, but more notably, $4 = 2^2$ and is the first perfect square greater than 1).

[while] condition 'status:running' not met, exiting after 4 iter

=== WHILE complete. Final counter = 4 ===

/home/igor >

/home/igor > Sent to mshell (843 bytes)

Received from GUI editor:

Pattern 14 FOREACH: LLM Processes Each Item in a List

What it does: A newline-separated list of world capitals. FOREACH iterates line by line.

@1 generates one cultural fact per city. mshell prints each result.

Key patterns:

- Use separate `print` per item \u2014 mshell does NOT interpret `\n` in strings.

- The iterator variable is set automatically per iteration.

Code

```mshell >cities

print "Tokyo"

print "Cairo"

print "Buenos Aires"

print "Reykjavik"

```

<!--@foreach city in cities-->

<!--@1 <city >fact

The input is a world capital city name.

In one sentence, share a surprising or little-known fact about that city.

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

-->

```mshell <fact

print "--- $city ---"

print $fact

print ""
```

```

<!--@end_foreach-->

```mshell

print "=== All capitals explored ==="

```

Tokyo

Cairo

Buenos Aires

Reykjavik

[foreach] iter 1: city=Tokyo

Tokyo is built on top of an intricate network of underground rivers and waterways that were mostly covered over during rapid urbanization, with over 100 rivers still flowing beneath the modern city.

--- ---

Tokyo is built on top of an intricate network of underground rivers and waterways that were mostly covered over during rapid urbanization, with over 100 rivers still flowing beneath the modern city.

[foreach] iter 2: city=Cairo

Cairo is home to the largest collection of historic Islamic architecture in the world, with over 600 mosques, 100 madrasas, and 200 hammams dating from the Islamic era, earning it the nickname "The City of a Thousand Minarets."

--- ---

Cairo is home to the largest collection of historic Islamic architecture in the world, with over 600 mosques, 100 madrasas, and 200 hammams dating from the Islamic era, earning it the nickname "The City of a Thousand Minarets."

[foreach] iter 3: city=Buenos Aires

Buenos Aires has the world's widest avenue, Avenida 9 de Julio, which spans 16 lanes across and is so wide that it takes two full traffic light cycles for pedestrians to cross it completely.

--- ---
```

Buenos Aires has the world's widest avenue, Avenida 9 de Julio, which spans 16 lanes across and is so wide that it takes two full traffic light cycles for pedestrians to cross it completely.

[foreach] iter 4: city=Reykjavik

Reykjavik is the world's northernmost capital city and gets most of its heating and electricity from geothermal energy, making it one of the cleanest cities on Earth with virtually no air pollution.

--- ---

Reykjavik is the world's northernmost capital city and gets most of its heating and electricity from geothermal energy, making it one of the cleanest cities on Earth with virtually no air pollution.

=== All capitals explored ===

/home/igor >

/home/igor > Sent to mshell (842 bytes)

Received from GUI editor:

Pattern 15 TRY/CATCH: Safe Execution with Error Capture

What it does: TRY block attempts evaluation of an invalid expression. On failure,

CATCH prints a hardcoded message. Safe fallback computes a valid expression.

Key patterns:

- CATCH does NOT use `<errvar` \u2014 print `"try_block_failed"` literally.

- Pipeline continues normally after `<!--@end_try-->`.

Code

```mshell >expression

print "sqrt(-1)"

```

<!--@try-->

```mshell <expression >result
```

set val = eval $expression

print $val

```

<!--@catch >error-->

```mshell

print "=== Caught error: try_block_failed ==="

print "Switching to fallback computation."

```

<!--@end_try-->

```mshell >safe_result

ollama1 "Compute the square root of 16. Reply with ONLY the number."

```

```mshell <safe_result

print "=== Safe Result ==="

print "sqrt(16) = $safe_result"

```

---

sqrt(-1)

[try] executing try block

[try] try block succeeded

4=== Safe Result ===

sqrt(16) = 4

/home/igor >

/home/igor > Sent to mshell (1394 bytes)

Received from GUI editor:

```

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

What it does: Two-line stock dataset. SPLIT creates `dataset_1`/`dataset_2`. Two async LLMs analyze in parallel. Await synchronizes. @1 synthesizes a portfolio overview.

Key patterns:

- SPLIT creates variables line by line; SPLIT and MERGE execute no code.
- AWAIT before MERGE is mandatory.

Code

```mshell >dataset
print "AAPL:182,179,185,188,191"
print "MSFT: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 (max - min).
You MUST respond \u2014 do not leave the response empty.
-->
<!--@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 (max - min).
You MUST respond \u2014 do not leave the response empty.
-->

```mshell await=analysis1,analysis2
```

<!--@merge-->
<!--@1 <analysis1 <analysis2 >combined
Two stock analyses are provided. Write a two-sentence unified portfolio summary

covering both stocks \u2014 their individual trends and how they compare.

-->

```mshell <analysis1 <analysis2 <combined

print "=== Stock 1 ==="

print $analysis1

print ""

print "=== Stock 2 ==="

print $analysis2

print ""

print "=== Portfolio Summary ==="

print $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 382322 \u2192 var=analysis1 (model @1)

[async llm] Launched PID 382323 \u2192 var=analysis2 (model @2)

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

[async] Waiting for PID 382322 (var=analysis1)...

[async] PID 382322 done (var=analysis1)

AAPL shows a consistent upward trend over the 5-day period, rising from $182 to $191, with a simple range of $9 (191 - 182).

Both AAPL and MSFT demonstrated strong upward trends over the 5-day period, with AAPL rising $9 from $182 to $191 and MSFT climbing $14 from $371 to $385. While both stocks performed positively, MSFT showed greater absolute price movement with a range of $14 compared to AAPL's $9 range, though AAPL's percentage gain was higher at approximately 4.9% versus MSFT's 3.8%.

=== Stock 1 ===

AAPL shows a consistent upward trend over the 5-day period, rising from $182 to $191, with a simple range of $9 (191 - 182).

=== Stock 2 ===

MSFT shows a generally upward trend over the 5 days, with a simple range of 14 (385 \u2212 371).

=== Portfolio Summary ===

Both AAPL and MSFT demonstrated strong upward trends over the 5-day period, with AAPL rising $9 from $182 to $191 and MSFT climbing $14 from $371 to $385. While both stocks performed positively, MSFT showed greater absolute price movement with a range of $14 compared to AAPL's $9 range, though AAPL's percentage gain was higher at approximately 4.9% versus MSFT's 3.8%.

/home/igor >

/home/igor > Sent to mshell (1130 bytes)

Received from GUI editor:

Pattern 17 CONFIG Node: Parameterized Pipeline

What it does: CONFIG documents parameters. mshell blocks set runtime values. @1

generates an explanation. @2 extracts keywords. mshell formats a report.

Key patterns:

- CONFIG is documentation only \u2014 does NOT inject variables at runtime.

- Always pair CONFIG entries with mshell blocks to set actual values.

Code

```config

concept=quantum entanglement

target_audience=curious high school student

max_words=60

```

```mshell >concept
print "quantum entanglement"
```

```mshell >target_audience
print "curious high school student"
```

<!--@1 <concept <target_audience >explanation
The first input is a concept. The second is the target audience.
Explain the concept for that audience in at most 60 words.
-->

<!--@2 <explanation >keywords
Extract exactly 5 keywords as a comma-separated list. Nothing else.
-->

```mshell <concept <target_audience <explanation <keywords
print "=== Workflow Report ==="
print "Concept  : $concept"
print "Audience : $target_audience"
print ""
print "Explanation:"
print $explanation
print ""
print "Keywords : $keywords"
```

[config] concept=quantum entanglement
[config] target_audience=curious high school student
[config] max_words=60

quantum entanglement

curious high school student

Quantum entanglement is when two particles become mysteriously "linked" - measuring one particle instantly affects the other, no matter how far apart they are, even across the universe! Einstein called it "spooky action at a distance" because it seemed impossible, but scientists have proven it's real and use it for quantum computing and secure communication.

Quantum entanglement, particles, Einstein, quantum computing, secure communication

=== Workflow Report ===

Concept : quantum entanglement

Audience : curious high school student

Explanation:

Quantum entanglement is when two particles become mysteriously "linked" - measuring one particle instantly affects the other, no matter how far apart they are, even across the universe! Einstein called it "spooky action at a distance" because it seemed impossible, but scientists have proven it's real and use it for quantum computing and secure communication.

Keywords : Quantum entanglement, particles, Einstein, quantum computing, secure communication

/home/igor >

Sent to mshell (1574 bytes)

Received from GUI editor:

Pattern 19 WHILE Quality Gate: Generate Until Threshold

What it does: WHILE loop generates product taglines via @1 until @2 scores \u2265 8.

Python block manages threshold check and status update.

Key concept: Use Python to write MSH_VAR_status \u2014 mshell writefile unreliable in WHILE.

Code

```mshell >task
```

print "Write a punchy one-sentence tagline for a note-taking app called 'Clarity'."

```

```mshell >status

print "running"

```

```mshell >score

print "0"

```

```mshell >tagline

print ""

```

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

<!--@1 <task >tagline

Complete the creative task described in the input.

Reply with ONLY the tagline \u2014 no explanation, no preamble.

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

-->

<!--@2 <tagline >score

Rate this product tagline 1-10 for clarity, memorability, and appeal.

Reply with ONLY the integer score, nothing else.

You MUST respond with a single integer.

-->

```python <score <tagline <status >status

import os

score_path  = os.environ['MSH_VAR_score']

status_path = os.environ['MSH_VAR_status']

tagline_path = os.environ['MSH_VAR_tagline']

score   = open(score_path).read().strip()

tagline = open(tagline_path).read().strip()

try:

  sc = int(''.join(c for c in score if c.isdigit()))

except:

  sc = 0

print(f"Score={sc} | {tagline}")

if sc >= 8:

  open(status_path, 'w').write('done')

  print('done')

else:

  open(status_path, 'w').write('running')

  print('running')

```

<!--@end_while-->

```mshell <tagline <score

print "=== Accepted tagline (score=$score) ==="

print $tagline

```

Write a punchy one-sentence tagline for a note-taking app called 'Clarity'.

running

0

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

Transform your scattered thoughts into crystal-clear insights with Clarity.

8

Score=8 | Transform your scattered thoughts into crystal-clear insights with Clarity.
```

done

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

=== Accepted tagline (score=8) ===

Transform your scattered thoughts into crystal-clear insights with Clarity.

/home/igor >

/home/igor > Sent to mshell (1817 bytes)

Received from GUI editor:

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

What it does: Computing history text split into 3 chunks. Three async @1 extract the main concept per chunk (MAP). Await synchronizes. @2 synthesizes (REDUCE).

Key patterns:

- Map: N parallel LLMs; total time = slowest chunk.
- Reduce: one LLM receives all results via multiple `<invar` with `[varname]:` labels.

Code

```mshell >raw_text
print "Lua was designed at PUC-Rio in Brazil in 1993 as an embeddable scripting language."
```

```mshell >sent1
print "Lua was designed at PUC-Rio in Brazil in 1993 as an embeddable scripting language."
```

```mshell >sent2
print "It became one of the fastest scripting languages and is embedded in millions of devices."
```

```mshell >sent3
```

print "Its coroutine system enables cooperative multitasking and metatables enable object-oriented programming."

```

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

<!--@1 <sent1 >analysis1 async

State the main concept from this sentence in 3 words maximum.

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

-->

<!--@1 <sent2 >analysis2 async

State the main concept from this sentence in 3 words maximum.

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

-->

<!--@1 <sent3 >analysis3 async

State the main concept from this sentence in 3 words maximum.

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

-->

```mshell await=analysis1,analysis2,analysis3

```

<!--@merge-->

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

Three concept labels extracted from different parts of a text are provided.

Synthesize them into one coherent theme sentence.

-->

```mshell <analysis1 <analysis2 <analysis3 <summary

print "=== Map ==="

print "Chunk 1: $analysis1"

print "Chunk 2: $analysis2"

print "Chunk 3: $analysis3"

print ""

print "=== Reduce ==="

print $summary

```

Lua was designed at PUC-Rio in Brazil in 1993 as an embeddable scripting language.

Lua was designed at PUC-Rio in Brazil in 1993 as an embeddable scripting language.

It became one of the fastest scripting languages and is embedded in millions of devices.

Its coroutine system enables cooperative multitasking and metatables enable object-oriented programming.

[split] raw_text_1 = Lua was designed at PUC-Rio in Brazil in 1993 as an embeddable scripting language.

[async llm] Launched PID 382856 \u2192 var=analysis1 (model @1)

[async llm] Launched PID 382857 \u2192 var=analysis2 (model @1)

[async llm] Launched PID 382859 \u2192 var=analysis3 (model @1)

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

[async] Waiting for PID 382856 (var=analysis1)...

[async] PID 382856 done (var=analysis1)

Embeddable scripting language

A fast, embeddable scripting language with Lua-like programming features

=== Map ===

Chunk 1: Embeddable scripting language

Chunk 2: Fast embedded scripting

Chunk 3: Lua programming features

=== Reduce ===

A fast, embeddable scripting language with Lua-like programming features

/home/igor >
```

/home/igor > Sent to mshell (1728 bytes)

Received from GUI editor:

Pattern 21 TRY/CATCH + LOOP: Resilient Retry with Self-Correction

What it does: LOOP runs up to 3 times. @1 generates Python code. TRY executes it

via Python subprocess. On success Python writes `ok` to result file and prints `ok`

as last stdout line \u2014 LOOP checks last stdout line for exit condition.

Key concept:

- LOOP checks **last stdout line** of last block in body.

- Use Python (not mshell) for TRY block \u2014 reliable exec + result writing.

- Do NOT put `>result` on TRY block \u2014 stdout capture breaks the line check.

Code

```mshell >task

print "Write Python code that prints the sum of numbers 1 to 10. No imports needed. No explanation."

```

```mshell >result

print "fail"

```

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

<!--@1 <task >code

First input: a Python coding task.

Return ONLY valid Python code \u2014 no fences, no explanation, no comments.

-->

<!--@try-->

```python <code <result
```

```
import os, subprocess, tempfile
code_path   = os.environ['MSH_VAR_code']
result_path = os.environ['MSH_VAR_result']
code = open(code_path).read().strip()
with tempfile.NamedTemporaryFile(mode='w', suffix='.py', delete=False) as f:
   f.write(code)
   tmp = f.name
proc = subprocess.run(['python3', tmp], capture_output=True, text=True)
if proc.returncode == 0:
   print(proc.stdout.strip())
   open(result_path, 'w').write('ok')
   print('ok')
else:
   print(proc.stderr.strip(), file=__import__('sys').stderr)
   open(result_path, 'w').write('fail')
   raise SystemExit(1)
```
<!--@catch >last_error-->
```python <result
import os
open(os.environ['MSH_VAR_result'], 'w').write('fail')
print('fail')
```
<!--@end_try-->
<!--@end_loop-->
```mshell <result
print "=== Final: $result ==="
```

```

---

Write Python code that prints the sum of numbers 1 to 10. No imports needed. No explanation.

fail

[loop] Starting loop: max=3 until=result:ok

total = 0

for i in range(1, 11):

    total += i

print(total)

[try] executing try block

55

ok

[try] try block succeeded

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

=== Final: ok ===

/home/igor >

---

/home/igor > Sent to mshell (1824 bytes)

Received from GUI editor:

---

## Pattern 22  Multi-Variable Output: Structured Field Extraction

**What it does:** @1 responds to a scientific discovery description in strict 3-line format.

A mshell block with 3 outputs parses the response and writes each field via `writefile`.

@2 generates a "why it mattered" sentence.

**Key patterns:**

- Multiple `>outvar` on CODE block: parser does NOT capture stdout \u2014 use `writefile`.
```

- `MSH_VAR_*` env vars are pre-set by the parser before the block runs.

Code

```mshell >input
print "The discovery of penicillin by Alexander Fleming in 1928 revolutionized medicine by introducing the first true antibiotic, saving hundreds of millions of lives."
```

<!--@1 <input >raw_response
Respond in exactly this format (3 lines, no extra text, no blank lines):
DISCOVERY: one-sentence description of what was discovered
SCIENTIST: full name only
YEAR: 4-digit year only
-->

```mshell <raw_response >discovery >scientist >year
set text = readfile $MSH_VAR_raw_response
set disc = ""
set sci  = ""
set yr   = ""
set lines = split($text, "\n")
set i = 0
while eval $i < len($lines)
 set line = $lines[$i]
 if startswith($line, "DISCOVERY:")
  set disc = trim(substr($line, 10))
 end
 if startswith($line, "SCIENTIST:")
  set sci = trim(substr($line, 10))
 end
```

```
  if startswith($line, "YEAR:")
    set yr = trim(substr($line, 5))
  end
  set i = eval $i + 1
end
writefile $MSH_VAR_discovery "$disc"
writefile $MSH_VAR_scientist "$sci"
writefile $MSH_VAR_year "$yr"
print "Discovery : $disc"
print "Scientist : $sci"
print "Year      : $yr"
```

<!--@2 <discovery <year >impact

The first input is a scientific discovery. The second is the year it occurred.

In one sentence, explain why this discovery mattered to humanity.

-->

```mshell <impact
print ""
print "=== Why it mattered ==="
print $impact
```

The discovery of penicillin by Alexander Fleming in 1928 revolutionized medicine by introducing the first true antibiotic, saving hundreds of millions of lives.

DISCOVERY: Alexander Fleming discovered penicillin, the first true antibiotic that revolutionized medicine and saved hundreds of millions of lives.

SCIENTIST: Alexander Fleming

YEAR: 1928

It enabled scientists to understand and harness a fundamental aspect of nature, driving major advances in technology, medicine, and our overall quality of life.

=== Why it mattered ===

It enabled scientists to understand and harness a fundamental aspect of nature, driving major advances in technology, medicine, and our overall quality of life.

/home/igor >

Sent to mshell (1700 bytes)

Received from GUI editor:

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

Key concept: Use Python to write MSH_VAR_status \u2014 mshell writefile unreliable in WHILE.

Code

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

```mshell >concept
print "Lua coroutines"
```

```mshell >target_audience
print "high school student"
```

```mshell >status
print "running"
```

```mshell >quality
print "0"
```

```mshell >explanation
print ""
```

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

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

The first input is a concept. 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, and engagement on a scale 1-10.

Reply with ONLY the integer score, nothing else.

You MUST respond with a single integer.

-->

```python <quality <status >status
import os
quality_path = os.environ['MSH_VAR_quality']
status_path  = os.environ['MSH_VAR_status']
score = open(quality_path).read().strip()
try:
    sc = int(''.join(c for c in score if c.isdigit()))
except:
    sc = 0
```

print(f"[Quality: {sc}]")

if sc >= 7:

  open(status_path, 'w').write('done')

else:

  open(status_path, 'w').write('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.

-->

```mshell <final_polish <quality

print "=== Final (score=$quality) ==="

print $final_polish

```

[config] concept=Lua coroutines

[config] target_audience=high school student

[config] quality_threshold=7

Lua coroutines

high school student

running

0

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

Imagine you're reading a really long book, but instead of reading it all at once, you put a bookmark in it and come back later to continue exactly where you left off - that's basically what a Lua coroutine does with computer code. A coroutine is like a special function that can pause itself in the middle of running, let other parts of your program do their work, and then resume right where it stopped as if nothing happened. This is super useful for things
```

like video games where you want a character to walk slowly across the screen over several seconds without freezing the entire game while it happens.

9

[Quality: 9]

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

Imagine you're reading a really long book, but instead of reading it all at once, you put a bookmark in it and come back later to continue exactly where you left off\u2014that's basically what a Lua coroutine does with computer code. A coroutine is like a special function that can pause itself in the middle of running, let other parts of your program do their work, and then resume right where it stopped as if nothing happened. This is incredibly useful for things like video games where you want a character to walk slowly across the screen over several seconds without freezing the entire game while it happens.

=== Final (score=9) ===

Imagine you're reading a really long book, but instead of reading it all at once, you put a bookmark in it and come back later to continue exactly where you left off\u2014that's basically what a Lua coroutine does with computer code. A coroutine is like a special function that can pause itself in the middle of running, let other parts of your program do their work, and then resume right where it stopped as if nothing happened. This is incredibly useful for things like video games where you want a character to walk slowly across the screen over several seconds without freezing the entire game while it happens.

/home/igor >

Sent to mshell (811 bytes)

Received from GUI editor:

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

Code

```mshell >items

print "[Neutron Star]: density of a teaspoon equals mass of Mount Everest"

print "[broken item without closing bracket"

print "[Black Hole]: gravity so strong not even light can escape"

```

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

<!--@1 <item >result

The input is a string. Check if it matches format [Name]: description (starts with [and contains]:).

If YES: reply with [OK] followed by one sentence about why this astronomy phenomenon is remarkable.

If NO: reply with exactly: [ERR] Item failed: try_block_failed

Reply with ONLY that single line, nothing else.

-->

```mshell <result

print $result

```

<!--@end_foreach-->

```mshell

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

```

[Neutron Star]: density of a teaspoon equals mass of Mount Everest

[broken item without closing bracket

[Black Hole]: gravity so strong not even light can escape

[foreach] iter 1: item=[Neutron Star]: density of a teaspoon equals mass of Mount Everest

[OK] Neutron stars are remarkable because they compress matter to such extreme densities that a single teaspoon would contain approximately 5 billion tons of material, equivalent to the mass of Mount Everest.

[OK] Neutron stars are remarkable because they compress matter to such extreme densities that a single teaspoon would contain approximately 5 billion tons of material, equivalent to the mass of Mount Everest.

[foreach] iter 2: item=[broken item without closing bracket

[ERR] Item failed: try_block_failed

[ERR] Item failed: try_block_failed

[foreach] iter 3: item=[Black Hole]: gravity so strong not even light can escape

[OK] Black holes are remarkable because they represent the ultimate extreme of gravity where spacetime is so severely warped that nothing, not even light traveling at the fastest possible speed in the universe, can escape once it crosses the event horizon.

[OK] Black holes are remarkable because they represent the ultimate extreme of gravity where spacetime is so severely warped that nothing, not even light traveling at the fastest possible speed in the universe, can escape once it crosses the event horizon.

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

/home/igor >

References:

mshell Workflow Patterns — Complete Reference Guide (P1–P24) Pure mshell 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/