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

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

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

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. Instead of writing separate scripts in different languages and manually wiring them together, mshell lets you define a workflow in a single Markdown document where each code block is a step in the pipeline.

This guide covers the **bash-only** variant: every computation, transformation, and formatting step is implemented with bash tools (`awk`, `sed`, `grep`, `tr`, `sort`, `printf`, `bc`, `wc`, etc.) — making workflows portable to any standard UNIX system without requiring language runtimes beyond bash itself.

Core Concepts

Supported Block Types (bash edition)

Block	Syntax	Role
Bash code	```` ```bash ``` ````	All computation, I/O, formatting
MShell code	```` ```mshell ``` ````	Native AI model calls (`ollama1/2/3`)
LLM directive	`<!--@N <in >out ... -->`	AI model call with prompt
Config	```` ```config ``` ````	Documentation block (no runtime effect)
WHILE	`<!--@while var:value-->`	Loop while var == value
FOREACH	`<!--@foreach item in list-->`	Iterate over lines of variable
TRY/CATCH	`<!--@try-->` / `<!--@catch >errvar-->`	Error isolation
SPLIT	`<!--@split var into N-->`	Visual marker; creates `var_1`, `var_2`, …

Block	Syntax	Role
MERGE	`<!--@merge-->`	Visual merge marker
LOOP	`<!--@loop max=N until=var:value-->`	Bounded retry loop

AI Model Integration

mshell supports up to **3 LLM models** simultaneously, addressed as `@1`, `@2`, and `@3`. Models can be Ollama-based local models or remote API endpoints (OpenAI, Claude, etc.), configured independently per model slot. Models run **synchronously** — each LLM block completes before the next block starts, guaranteeing that output variables are available to all subsequent blocks.

Variable System

Data flows between blocks through a **file-based context system**:

```
bash >varname          # produce output into a named variable
bash <varname          # consume variable from a previous block
bash <var1 <var2       # multiple inputs supported
bash >out1 >out2       # multiple outputs: must write to $MSH_VAR_* directly
```

Variables persist as files in `/tmp/mshell_ctx_PID/` and are accessible across all blocks via the `MSH_VAR_varname` environment variable.

Multi-Input LLM Prompts

With **one** `<invar`: variable contents are injected without a label.

With **multiple** `<invar`: each variable is injected with a `[varname]:` label:

```
[topic]:
the CAP theorem
[audience]:
junior developer
```

In prompts with multiple variables, refer to them as “The first input”, “The second input”, or by their `[varname]:` labels.

Conditional Execution

Any block can be conditionally executed based on a variable value:

```
bash <route if=route:MATH
```

The block runs only if the variable `route` contains the value `MATH`.

Bash Toolbox Quick Reference

Common bash tools used throughout these patterns:

Tool	Use case
`awk`	Numeric computation, field extraction, statistics
`sed`	Text substitution, line filtering
`grep -oP` / `grep -oE`	Regex field extraction
`sort -t, -k2 -rn`	CSV numeric sort
`tr '[:lower:]' '[:upper:]'`	Case conversion
`tr -d '[:space:]'`	Strip whitespace (LLM score cleaning)
`wc -l` / `wc -w` / `wc -c`	Line/word/char counting
`printf '%x'`	Integer to hex conversion
`bc`	Arbitrary precision arithmetic
`rev`	Reverse a string
`fold -w N`	Word-wrap for report formatting
`seq 1 N`	Integer range generation

Node Syntax Reference

Node	Syntax	Semantics
WHILE	`<!--@while var:value-->` … `<!--@end_while-->`	Loop while var == value
FOREACH	`<!--@foreach item in listvar-->` … `<!--@end_foreach-->`	Iterate over lines
TRY/CATCH	`<!--@try-->` … `<!--@catch >errvar-->` … `<!--@end_try-->`	Error isolation
LOOP	`<!--@loop max=N until=var:val-->` … `<!--@end_loop-->`	Bounded retry
SPLIT	`<!--@split var into N-->`	Creates `var_1`, `var_2`, …
MERGE	`<!--@merge-->`	Visual reduce marker
CONFIG	```` ```config … ``` ````	Parameter documentation

Critical rules:

- **FOREACH**: list must be created with `printf "a\nb\nc"` (one item per line).
- **WHILE**: condition is checked before body; body must modify the condition variable.
- **CONFIG**: does not inject variables at runtime — always pair with bash `echo` blocks.
- **CATCH `>errvar`**: variable receives the literal string `"try_block_failed"`. Do **NOT** use `<errvar` inside the CATCH block.
- **Multiple `>outvar` on CODE block**: block must write to `$MSH_VAR_*` files directly.
- **`until=var:VALUE` in LOOP**: checks the **last non-empty line** of the variable. Always normalize LLM output with a bash block (`head -1 | tr -d '[:space:]'`) before the condition variable is used by `until=`.

Patterns Part I (p1–p12)

Pattern 1 — Linear Data Pipeline (7 Bash Stages)

What it does: Demonstrates sequential data flow through 7 bash stages. A value produced in Stage 1 is transformed through doubling, hex conversion, string reversal, statistics computation, uppercasing, and finally ASCII-frame formatting. Each stage reads the output of the previous block and passes its own output to the next.

Use case: Multi-stage ETL pipelines where each bash block handles a specific transformation — entirely without external language runtimes.

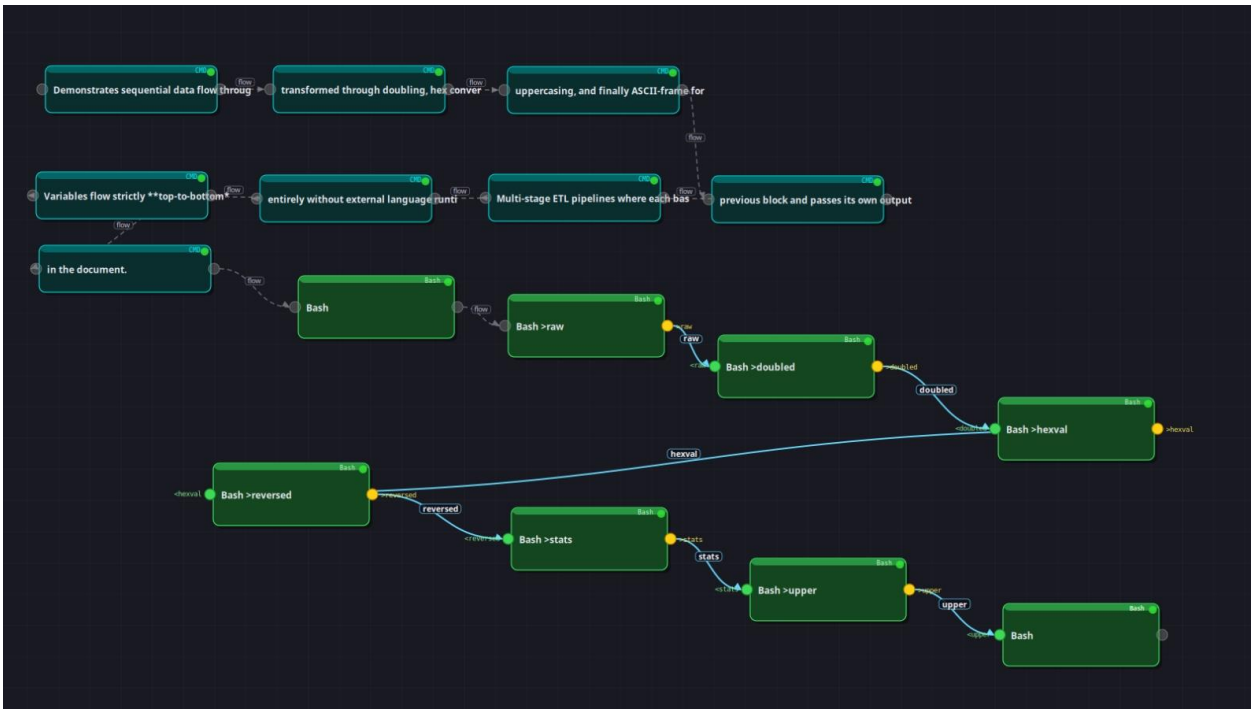
Key concept: Variables flow strictly **top-to-bottom**. Each producer must appear before its consumers in the document.

Flow diagram:

```
bash >raw         (generate integer 42)
bash <raw >doubled       (×2, label)
bash <doubled >hexval    (printf '%x')
bash <hexval >reversed   (rev)
bash <reversed >stats    (wc -c / wc -w)
bash <stats >upper       (tr to uppercase)
bash <upper              (ASCII-frame report)
```

Code:

```
bash >raw echo "42"

bash <raw >doubled val=$(cat "$MSH_VAR_raw") result=$((val * 2)) echo
"value=${val} doubled=${result}"

bash <doubled >hexval text=$(cat "$MSH_VAR_doubled") num=$(echo "$text" |
grep -oP 'doubled=\K[0-9]+') hex=$(printf '%x' "$num") echo
"hex_of_doubled=${hex}"

bash <hexval >reversed line=$(cat "$MSH_VAR_hexval") hex=$(echo "$line" |
grep -oP 'hex_of_doubled=\K\S+') rev_hex=$(echo "$hex" | rev) echo
"reversed_hex=${rev_hex}"

bash <reversed >stats line=$(cat "$MSH_VAR_reversed") chars=$(echo "$line" |
wc -c) words=$(echo "$line" | wc -w) echo "chars=${chars} words=${words}
data=${line}"

bash <stats >upper cat "$MSH_VAR_stats" | tr '[:lower:]' '[:upper:]'

bash <upper line=$(cat "$MSH_VAR_upper") border=$(printf '%0.s=' $(seq 1
${#line})) echo "$border" echo "$line" echo "$border" echo "=== Pipeline
complete: 7 bash stages ==="
```



Pattern 2 — LLM in the Middle

What it does: A bash block generates structured server log data. LLM @1 receives the log and produces a one-paragraph incident analysis. A second bash block post-processes the LLM response: it counts sentences, extracts numbers, and prints a formatted digest using `grep`, `awk`, `tr`, and `wc`.

Use case: Automated data analysis pipelines where you want a natural language interpretation of structured data, followed by further extraction of metadata from the prose response.

Key concept: LLM directive `<!--@1 <data >result ... -->` takes input from a variable, sends it as part of the prompt, and stores the model response in an output variable accessible to subsequent blocks.

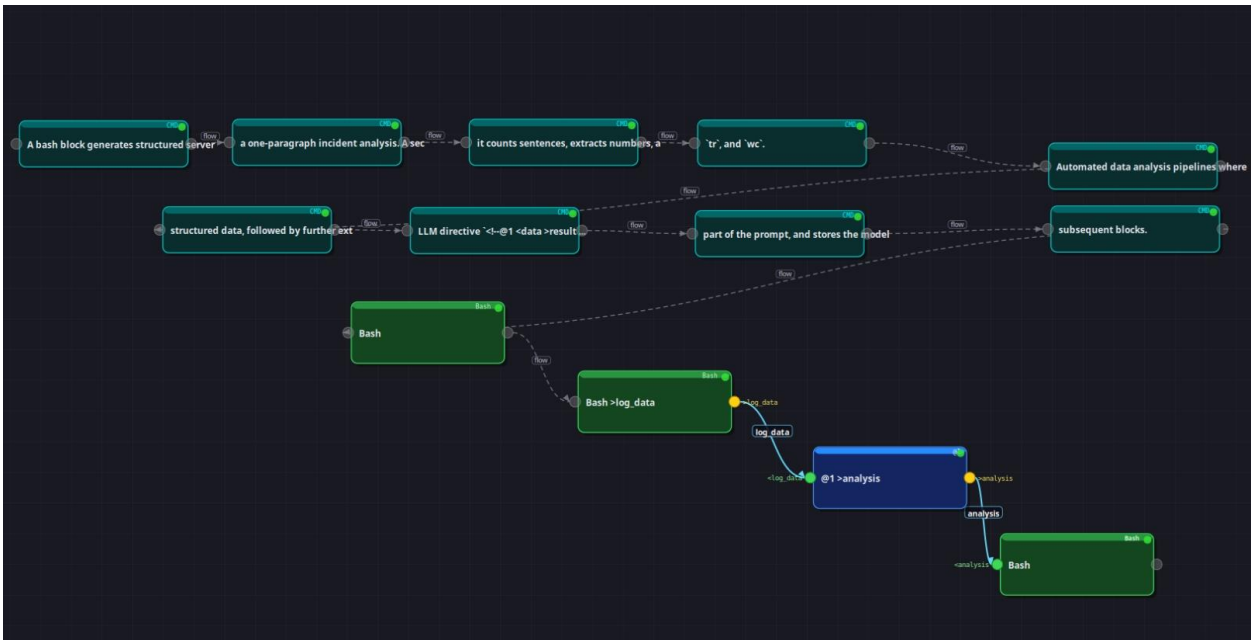
Flow diagram:

```
bash >log_data          (structured server log summary)
@1 <log_data >analysis  (LLM: incident analysis paragraph)
bash <analysis          (extract stats, print digest)
```

Code:

```
bash >log_data cat <<'EOF' Server log summary (last 1h): 200 OK        : 8420
requests  avg=42ms 301 Redirect  : 312  requests  avg=5ms 400 Bad Req   : 87
requests  avg=12ms 404 Not Found : 203  requests  avg=8ms 500 Error     : 34
requests  avg=890ms 503 Unavail   : 11   requests  avg=2100ms EOF
```

```
bash <analysis text=$(cat "$MSH_VAR_analysis") sentences=$(echo "$text" |
grep -oE '[^.!?]+[.!?]' | wc -l) numbers=$(echo "$text" | grep -oE '[0-9]+' |
tr '\n' ',' | sed 's/,$//') echo "=== LLM Incident Analysis ===" echo "$text"
echo "" echo "--- Digest ---" echo "Sentences : $sentences" echo "Numbers
mentioned : $numbers" echo "Word count : $(echo "$text" | wc -w)"
```



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

What it does: A single CSV dataset of city temperatures is produced once and consumed independently by three downstream blocks: bash computes statistics with `awk`, bash sorts cities by temperature, and LLM @1 writes a weather narrative.

Use case: Parallel analysis of the same dataset from different perspectives without duplicating data.

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

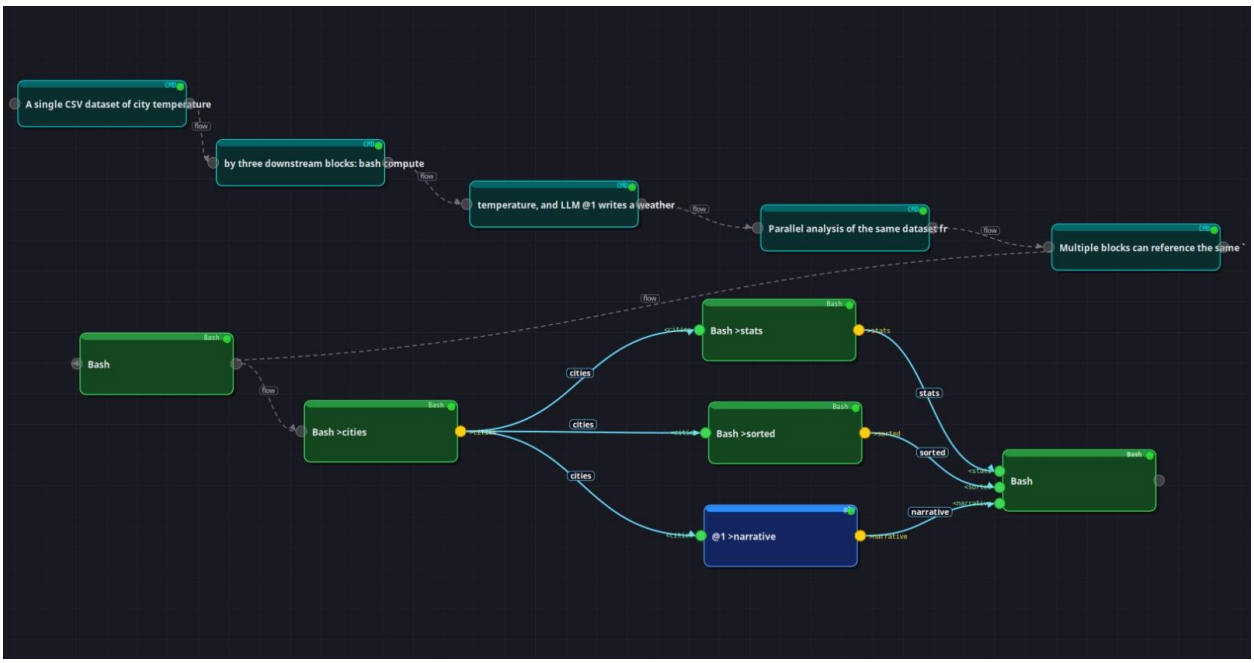
Flow diagram:

```
bash >cities             (CSV: city,temp_celsius)
bash <cities >stats      (awk: min/max/avg)
bash <cities >sorted     (sort descending)
@1  <cities >narrative   (LLM: weather sentence)
bash <stats <sorted <narrative  (unified report)
```

Code:

```
bash >cities printf "London,12\nParis,18\nMoscow,-
3\nDubai,38\nTokyo,22\nNewYork,9\nSydney,25\n"
```

```
bash <cities >stats data=$(cat "$MSH_VAR_cities") result=$(echo "$data" | awk
-F',' '   BEGIN { min=9999; max=-9999; sum=0; n=0 }   { t=$2+0; if(t<min)
min=t; if(t>max) max=t; sum+=t; n++ }   END { printf "count=%d min=%d max=%d
avg=%.1f", n, min, max, sum/n } ') echo "$result"

bash <cities >sorted echo "=== Cities by temperature (desc) ===" cat
"$MSH_VAR_cities" | sort -t',' -k2 -rn | \   awk -F',' '{ printf "  %-12s
%3d°C\n", $1, $2 }'

bash <stats <sorted <narrative echo "=== Statistics ===" && cat
"$MSH_VAR_stats" echo "" && cat "$MSH_VAR_sorted" echo "" && echo "=== AI
Narrative ===" && cat "$MSH_VAR_narrative"
```



Pattern 4 — LLM Code Generation → Execute via Variable

What it does: A bash block stores a shell task description. LLM @1 generates a **pure bash script** that solves it. A second bash block inspects and executes the script via `bash "$MSH_VAR_code"`.

Use case: Automated bash script generation workflows — no copy-paste needed.

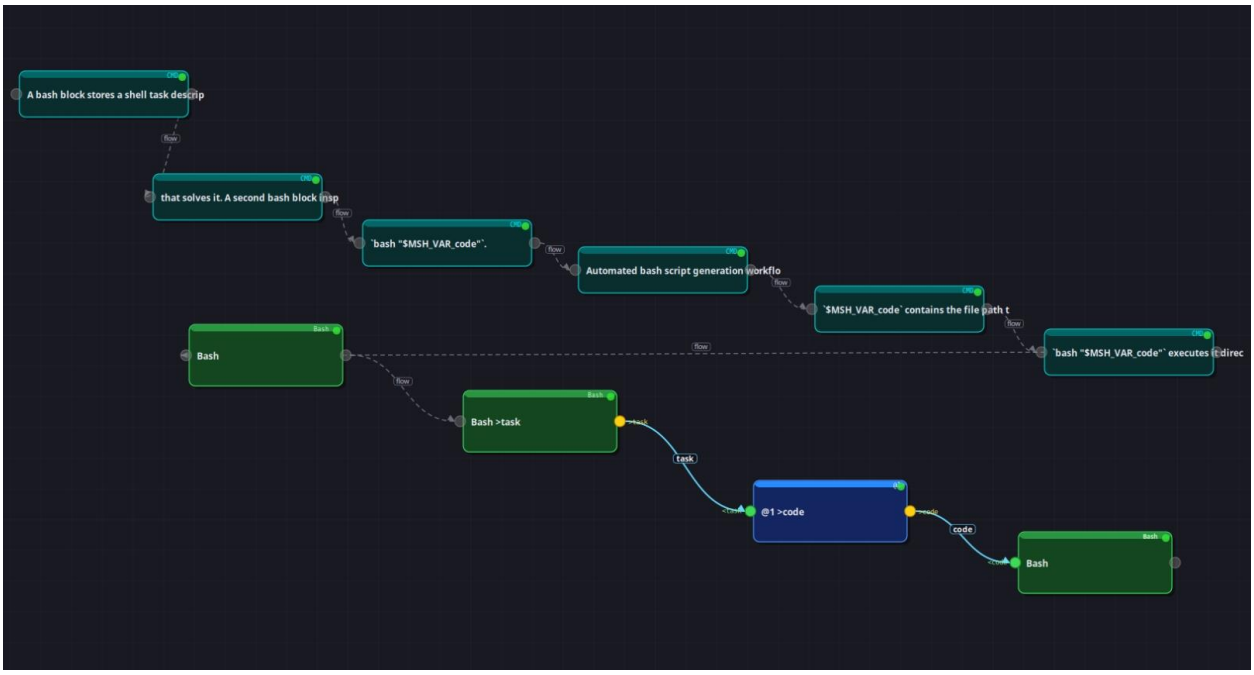
Key concept: `$MSH_VAR_code` contains the file path to the generated script. `bash "$MSH_VAR_code"` executes it directly.

Flow diagram:

```
bash >task
@1 <task >code    (LLM: generate pure bash script, no fences)
bash <code        (inspect + execute)
```

Code:

```
bash >task echo "Write a bash script that takes the sentence 'the cat sat on
the mat and the cat sat' and prints each word that appears more than once
along with its count, sorted by count descending."

bash <code echo "=== Generated bash script ===" cat "$MSH_VAR_code" echo ""
echo "=== Executing ===" bash "$MSH_VAR_code"
```



Pattern 5 — Two-LLM Review Chain

What it does: LLM @1 generates a bash script. LLM @2 reviews it. LLM @1 receives both script and review, produces an improved version. A final bash block executes it.

Use case: Automated bash script quality improvement — generate, review, and refine in a single pipeline.

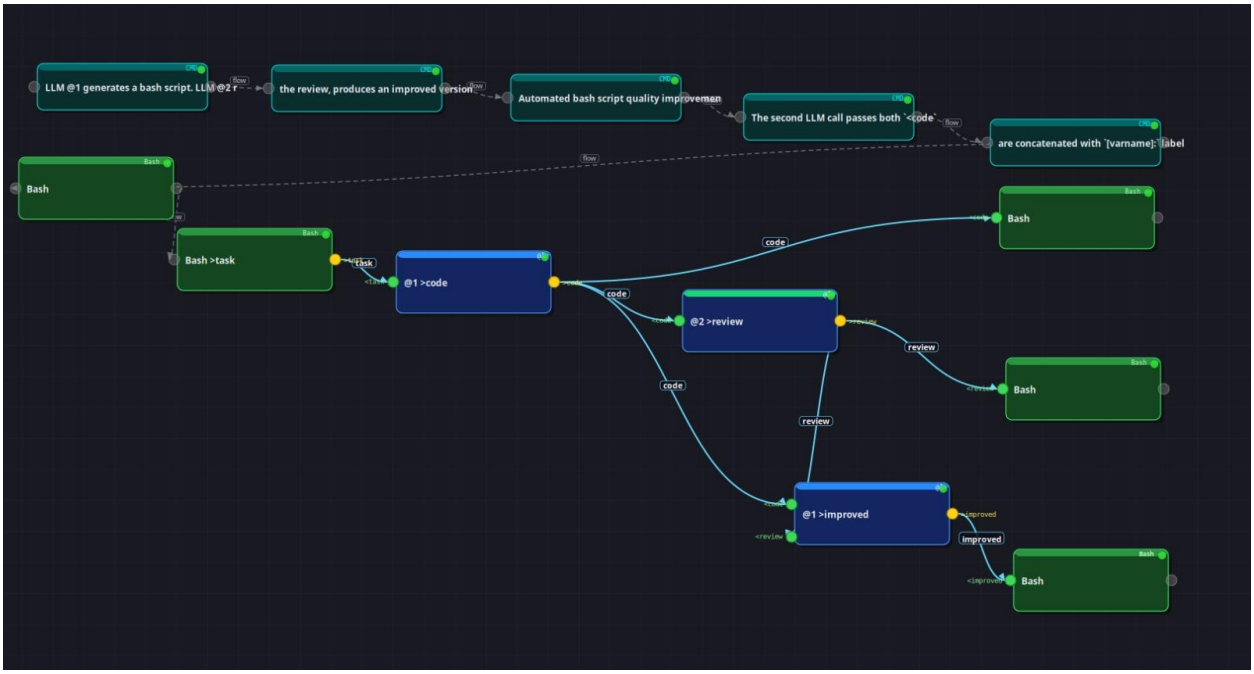
Key concept: The second LLM call passes both `<code` and `<review` as inputs. Multiple input variables are concatenated with `[varname]:` labels automatically.

Flow diagram:

```
bash >task
@1 <task >code           (generate bash script)
bash <code               (print)
@2 <code >review         (review for correctness)
bash <review             (print)
@1 <code <review >improved   (improve)
bash <improved           (print + execute)
```

Code:

```
bash >task echo "Write a bash function is_valid_ipv4 that takes one argument
and returns 0 (true) if it is a valid IPv4 address, 1 (false) otherwise. Test
it on: 192.168.1.1, 256.0.0.1, 10.0.0, 0.0.0.0, and 999.999.999.999"

bash <code echo "=== Model 1 generated ===" cat "$MSH_VAR_code"

bash <review echo "=== Model 2 review ===" cat "$MSH_VAR_review"

bash <improved echo "=== Final improved script ===" cat "$MSH_VAR_improved"
echo "" && echo "=== Executing ===" bash "$MSH_VAR_improved"
```



Pattern 6 — Parallel 3-Model Query

What it does: The same system design question is sent to all three LLM models independently. A final bash block collects all three responses and prints a comparison panel with word counts.

Use case: Comparing answers across models, collecting ensemble responses, or generating diverse perspectives on the same topic.

Key concept: Three LLM directive blocks with `@1`, `@2`, `@3` all read the same `<question` and write to separate `>ans1`, `>ans2`, `>ans3`.

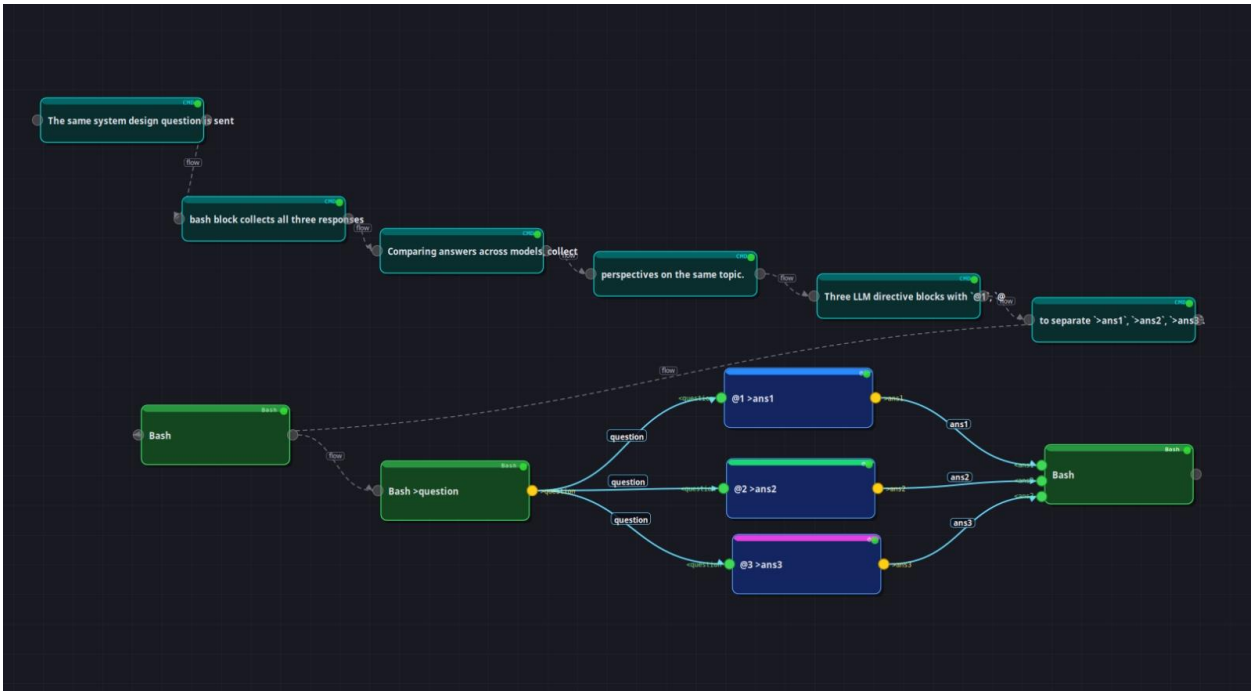
Flow diagram:

```
bash >question
@1 <question >ans1
@2 <question >ans2
```

```
@3 <question >ans3
bash <ans1 <ans2 <ans3   (comparison panel)
```

Code:

```
bash >question echo "You are designing a URL shortener service expecting 100M requests/day. In one sentence, name the single most critical architectural decision and why."

bash <ans1 <ans2 <ans3 w1=$(cat "$MSH_VAR_ans1" | wc -w) w2=$(cat "$MSH_VAR_ans2" | wc -w) w3=$(cat "$MSH_VAR_ans3" | wc -w) echo "▶ Model @1 (${w1}w): $(cat $MSH_VAR_ans1)" echo "" echo "▶ Model @2 (${w2}w): $(cat $MSH_VAR_ans2)" echo "" echo "▶ Model @3 (${w3}w): $(cat $MSH_VAR_ans3)"
```



Pattern 7 — Evaluator-Optimizer Loop

What it does: LLM @1 generates a bash script. LLM @2 returns `ACCEPTED` or `REJECTED`. A bash normalization block strips the verdict to its first line so the `until=` condition matches reliably. The loop repeats until acceptance or max iterations. Final script is executed.

Use case: Automated iterative refinement with quality gates.

Key concept: `<!--@loop max=N until=verdict:ACCEPTED-->` wraps the generator-evaluator pair. Exits early on acceptance or after N iterations.

Critical rule: `until=verdict:ACCEPTED` checks the **last non-empty line** of the variable. LLM @2 must return `ACCEPTED` or `REJECTED` as a **single word, nothing else**. A bash

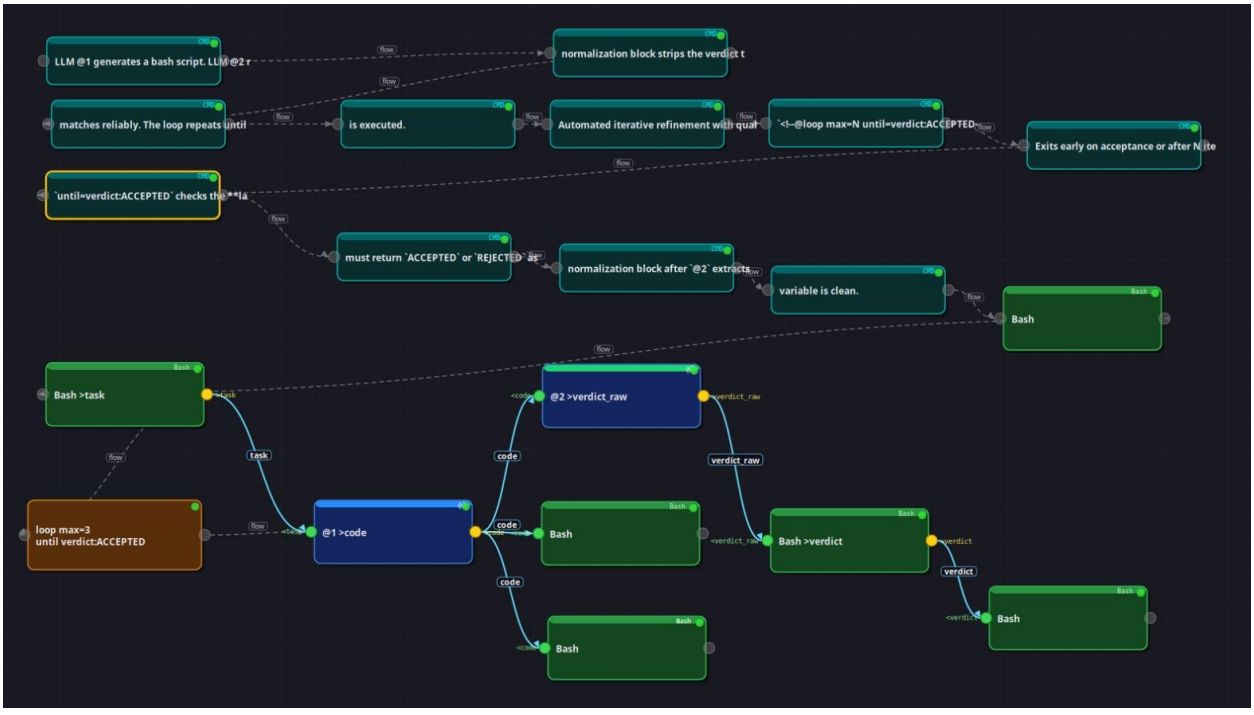
normalization block after @2 extracts only the first line to ensure the condition variable is clean.

Flow diagram:

```
bash >task
[LOOP max=3 until=verdict:ACCEPTED]
  @1 <task >code            (generate bash script)
  bash <code                (print)
  @2 <code >verdict_raw     (ACCEPTED or REJECTED, one word only)
  bash <verdict_raw >verdict (normalize: head -1, strip whitespace)
  bash <verdict             (print)
[END_LOOP]
bash <code                  (execute accepted script)
```

Code:

```
bash >task echo "Write a bash function seconds_to_human that converts seconds to '2h 5m 30s'. Omit zero units. Test with: 0, 59, 3661, 86400, 90061."

bash <code echo "=== Generated script ===" cat "$MSH_VAR_code"

bash <verdict_raw >verdict head -1 "$MSH_VAR_verdict_raw" | tr -d '[:space:]'

bash <verdict echo "=== Verdict ===" && cat "$MSH_VAR_verdict" && echo ""

bash <code echo "=== Final accepted script — executing ===" bash "$MSH_VAR_code"
```



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

What it does: Replaces the C/Python/Rust multi-language pipeline with a fully bash equivalent: bash generates data with `awk`, bash computes statistics, LLM @1 analyzes, LLM @2 summarizes, bash formats the final report.

Use case: Complex data processing workflows requiring computational processing and natural language intelligence — no external language runtimes.

Key concept: Language blocks and LLM blocks are interchangeable pipeline stages.

Flow diagram:

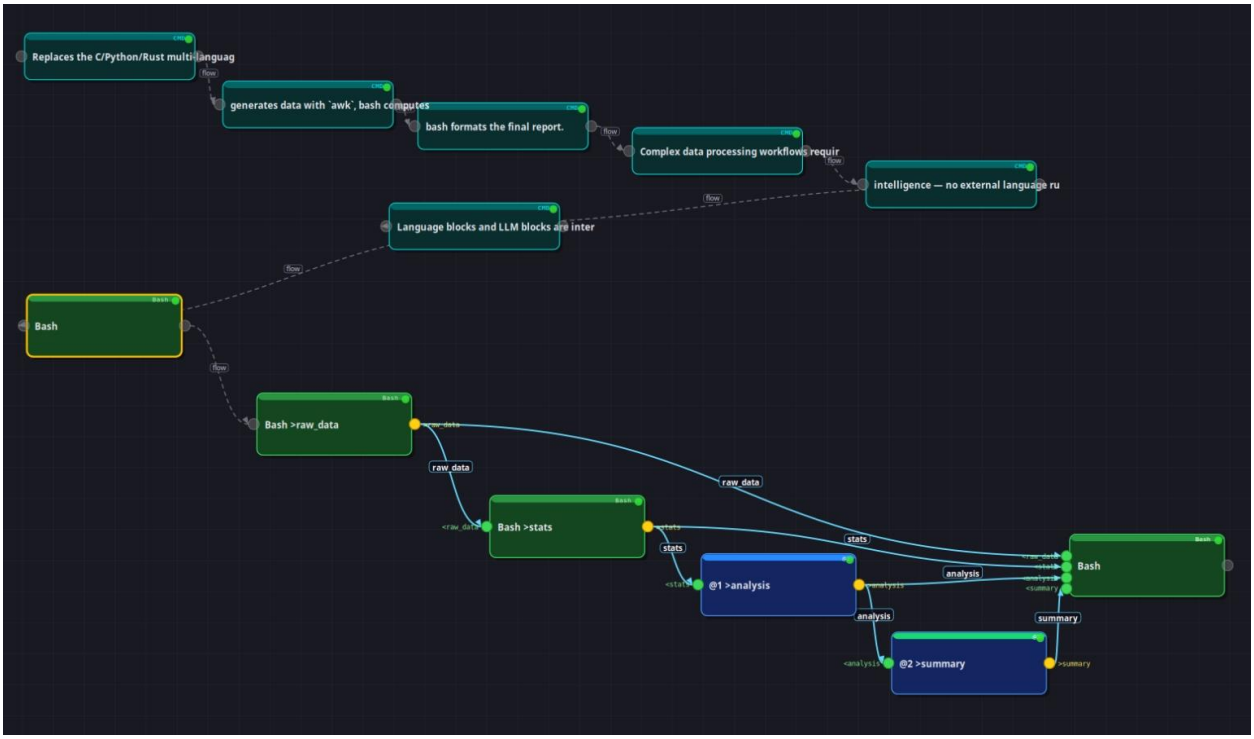
```
bash >raw_data   (awk: 10 pseudo-random numbers)
bash <raw_data >stats    (awk: stats)
@1 <stats >analysis      (LLM: distribution insight)
@2 <analysis >summary    (LLM: 5-word tweet)
bash <raw_data <stats <analysis <summary   (report)
```

Code:

```
bash >raw_data awk 'BEGIN { srand(42); for(i=1;i<=10;i++) printf "%d%s",
int(rand()*100), (i<10?",":"\n") }'
```

```
bash <raw_data >stats echo "$(cat $MSH_VAR_raw_data)" | tr ',' '\n' | awk '
BEGIN { min=9999; max=-9999; sum=0; n=0 }   { v=$1+0; if(v<min)min=v;
if(v>max)max=v; sum+=v; n++ }   END { printf "count=%d sum=%d min=%d max=%d
mean=%.2f\n", n,sum,min,max,sum/n } '
```

```
bash <raw_data <stats <analysis <summary echo "[raw_data] : $(cat
$MSH_VAR_raw_data)" echo "[stats]    : $(cat $MSH_VAR_stats)" echo
"[analysis] : $(cat $MSH_VAR_analysis)" echo "[summary]  : $(cat
$MSH_VAR_summary)"
```



Pattern 9 — Routing: LLM Classifies → Conditional Branch Executes

What it does: LLM @1 classifies input into `SYSINFO`, `TEXTPROC`, or `MATHCALC`. Only the matching bash branch executes. Three test cases demonstrated.

Use case: Dynamic task routing — appropriate handler depends on the nature of the input.

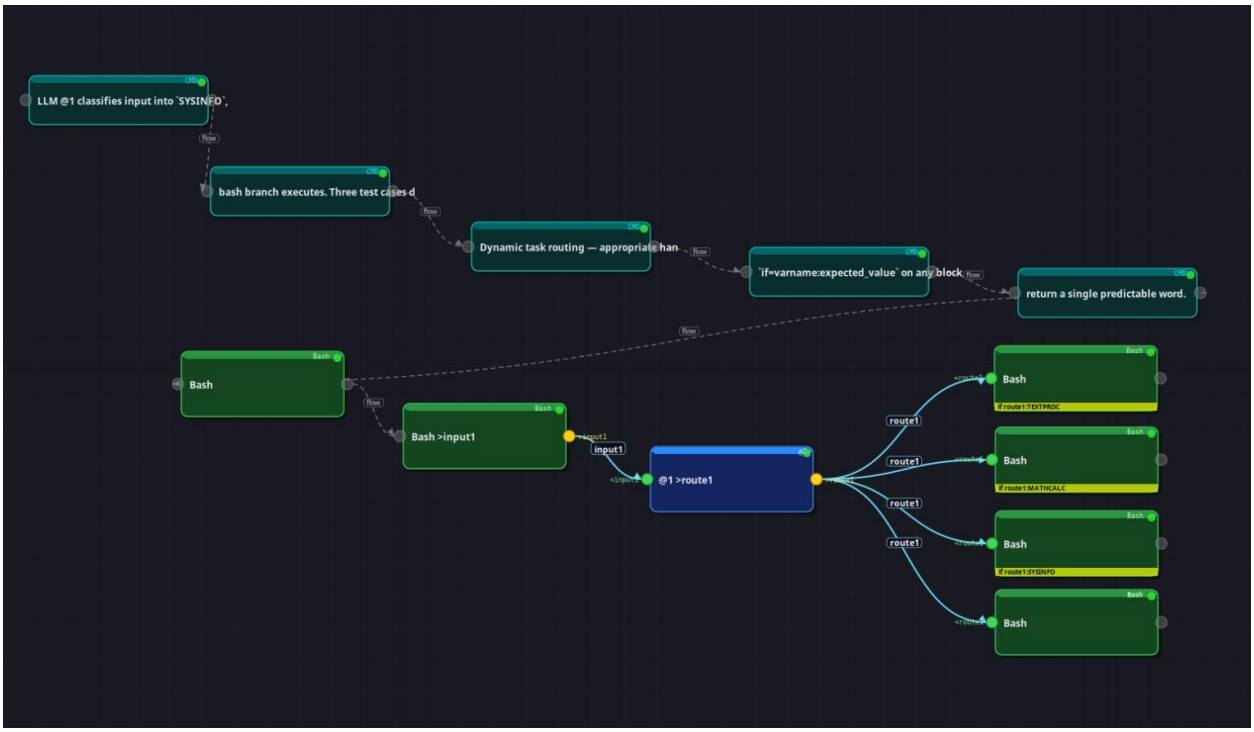
Key concept: `if=varname:expected_value` on any block makes it conditional. The LLM classifier must return a single predictable word.

Flow diagram:

```
bash >inputN
@1 <inputN >routeN        (classify)
bash <routeN if=routeN:SYSINFO    (system info)
bash <routeN if=routeN:TEXTPROC   (text analysis)
bash <routeN if=routeN:MATHCALC   (arithmetic)
bash <routeN                      (print classification)
```

Code: *(three test cases — see full pattern-09.md for complete code)*

```
bash >input1 echo "What is the current system load and disk usage?"

bash <route1 if=route1:SYSINFO echo "=== SYSINFO ===" && uname -s && uptime

bash <route1 echo "→ Classified as: $(cat $MSH_VAR_route1)"
```



Pattern 10 — Full Pipeline: All Patterns Combined

What it does: Bash generates Fibonacci numbers, bash computes statistics, LLM @1 analyzes, LLM @2 writes a poem, bash collects both, LLM @1 synthesizes, bash formats with decorative ASCII frame.

Use case: A reference example showing how all building blocks compose naturally.

Key concept: There is no special "combine patterns" syntax — patterns compose because every block reads from named variables. The document structure defines the execution graph.

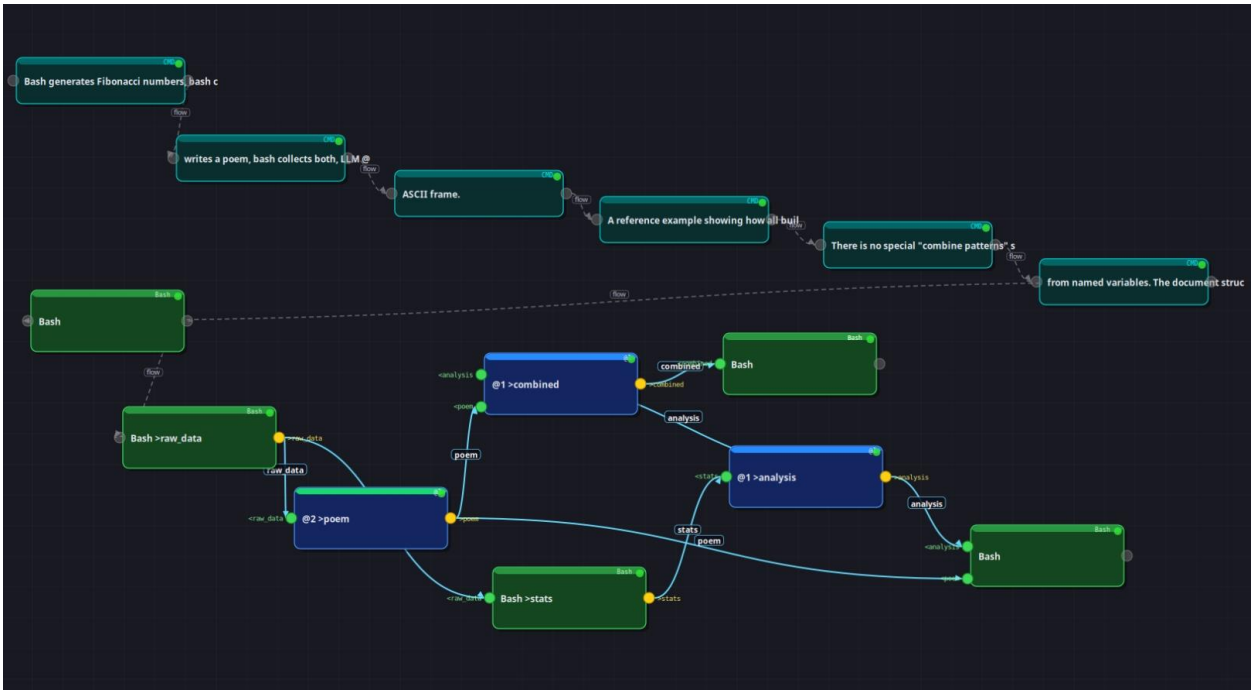
Flow diagram:

```
bash >raw_data   (Fibonacci via bash loop)
bash <raw_data >stats    (awk: statistics)
@1 <stats >analysis      (LLM: insight)
@2 <raw_data >poem       (LLM: 2-line poem)
bash <analysis <poem     (collect)
@1 <analysis <poem >combined   (synthesize)
bash <combined           (ASCII-framed output)
```

Code:

```
bash >raw_data a=0; b=1; result="" for i in $(seq 1 10); do
result="${result}${b} "   tmp=$((a+b)); a=$b; b=$tmp done echo "${result% }"
| tr ' ' ','
```

```
bash <raw_data >stats echo "$(cat $MSH_VAR_raw_data)" | tr ',' '\n' | awk '
BEGIN { sum=0; max=0; n=0; prev=0 }   { v=$1+0; sum+=v; n++; if(v>max)max=v;
if(NR>1&&prev>0)ratio=v/prev; prev=v }   END { printf "count=%d sum=%d max=%d
mean=%.1f last_ratio=%.4f\n",n,sum,max,sum/n,ratio } '

bash <analysis <poem echo "Analysis: $(cat $MSH_VAR_analysis)" && echo "" &&
echo "Poem:" && cat "$MSH_VAR_poem"

bash <combined text=$(cat "$MSH_VAR_combined") border=$(printf '%0.s=' $(seq
1 60)) echo "$border" && echo "$text" | fold -w 60 && echo "$border"
```



Pattern 11 — MShell Node with Multiple Models

What it does: Uses native `mshell` blocks with `ollama1`/`ollama2` alongside bash. Bash sets input variables, mshell calls generate explanation and keywords, bash parses and formats the output.

Use case: Workflows where mshell's native AI commands are preferable to LLM directives.

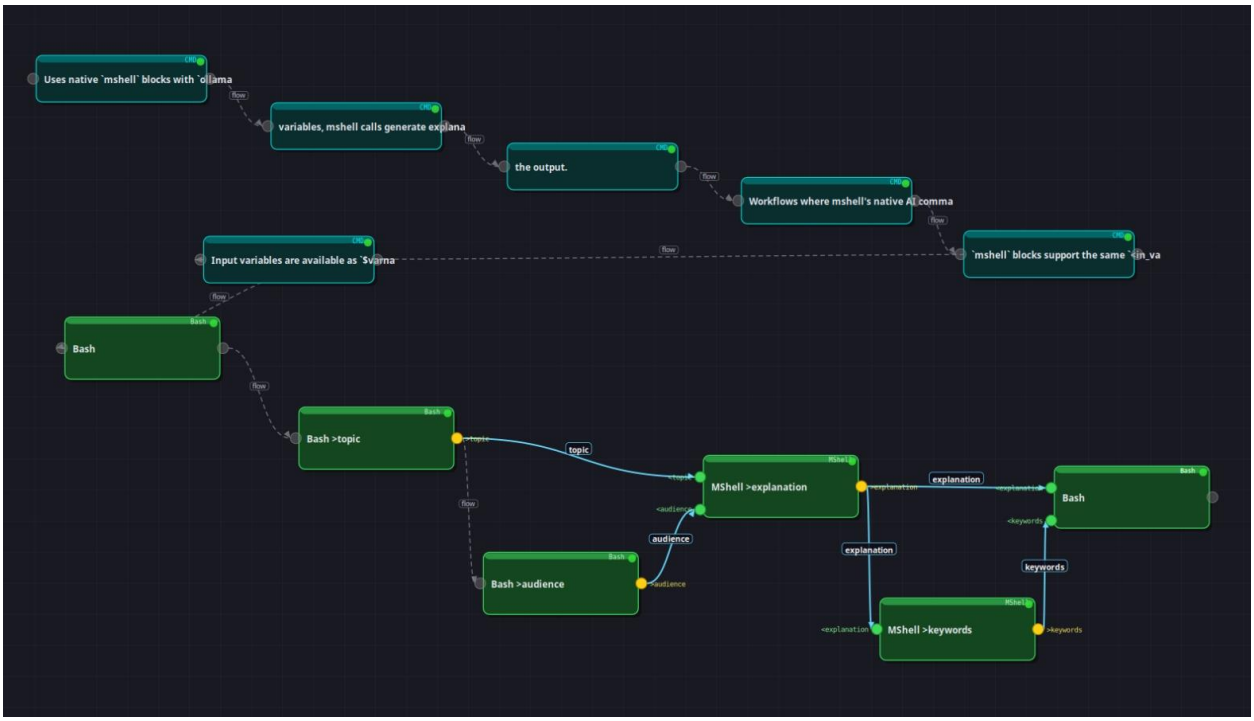
Key concept: `mshell` blocks support the same `<in_vars` and `>out_var` system as all other blocks. Input variables available as `$varname` inside mshell.

Flow diagram:

```
bash >topic >audience
mshell <topic <audience >explanation   (ollama1)
mshell <explanation >keywords          (ollama2)
bash <explanation <keywords            (parse + digest)
```

Code:

```
bash >topic echo "distributed systems"

bash >audience echo "senior engineer"

mshell <topic <audience >explanation ollama1 "Explain $topic in one sentence
tailored for a $audience"

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

bash <explanation <keywords echo "Topic    : $(cat $MSH_VAR_topic)" echo
"Audience : $(cat $MSH_VAR_audience)" echo "Explanation: $(cat
$MSH_VAR_explanation)" echo "Keywords : $(cat $MSH_VAR_keywords)"
```



Pattern 12 — Sequential 3-Model Query + Synthesis

What it does: Three LLM models answer the same question sequentially from three different perspectives (beginner, analogy, technical). LLM @1 synthesizes all three into one sentence. Bash formats the comparison panel.

Use case: High-quality ensemble answer generation — each model contributes a distinct perspective, then a synthesis pass unifies them.

Key concept: Three sequential LLM calls write to independent output variables. All three are available to the synthesis call and the final bash block.

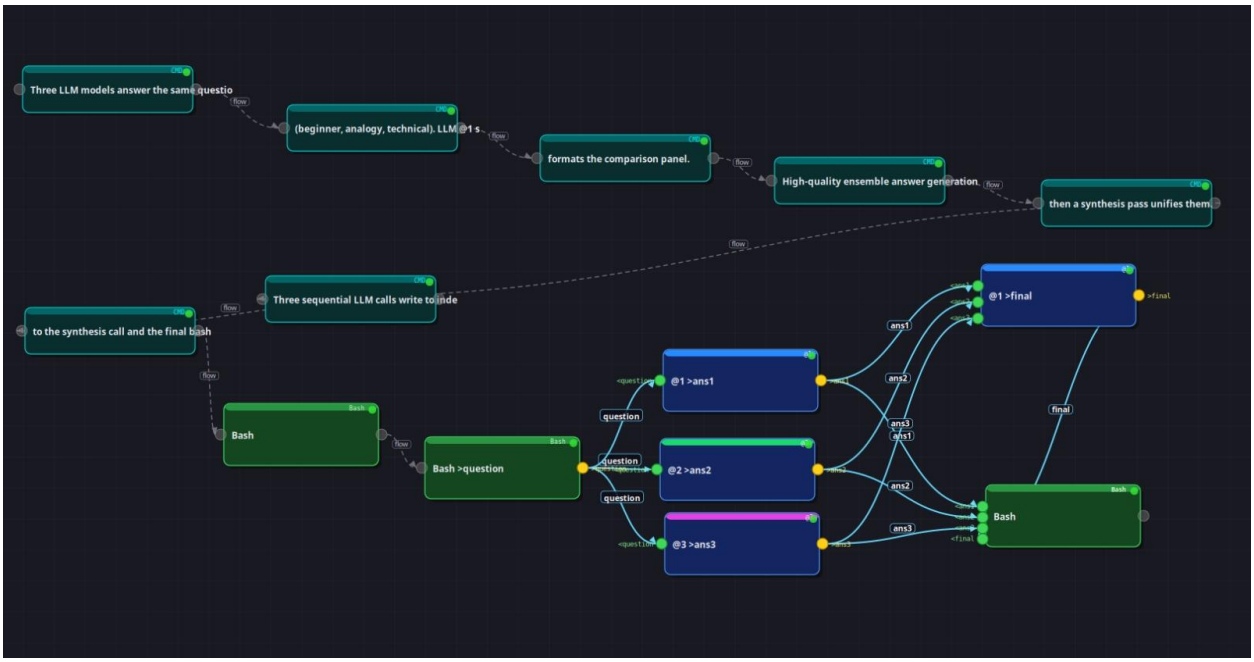
Flow diagram:

```
bash >question
@1 <question >ans1   (beginner perspective)
@2 <question >ans2   (real-world analogy)
@3 <question >ans3   (technical UNIX terms)
@1 <ans1 <ans2 <ans3 >final   (synthesize)
bash <ans1 <ans2 <ans3 <final (comparison panel)
```

Code:

```
bash >question echo "What is a pipe in bash and why is it one of the most
powerful features of UNIX?"
```

```
bash <ans1 <ans2 <ans3 <final echo "Beginner : $(cat $MSH_VAR_ans1)" echo ""
echo "Analogy  : $(cat $MSH_VAR_ans2)" echo "" echo "Technical: $(cat
$MSH_VAR_ans3)" echo "" echo "SYNTHESIS: $(cat $MSH_VAR_final)"
```



Patterns Part II (p13–p24)

Pattern 13 — WHILE Loop: Iterative Counter with LLM Commentary

What it does: Initializes `status=running` and `counter=0`. WHILE loop runs while `status == running`. Each iteration: bash increments counter, writes via `$MSH_VAR_*`. LLM gives one UNIX/bash fact about the current number. At counter≥3 status becomes `done`.

Use case: Any iterative processing where the exit condition depends on accumulated state.

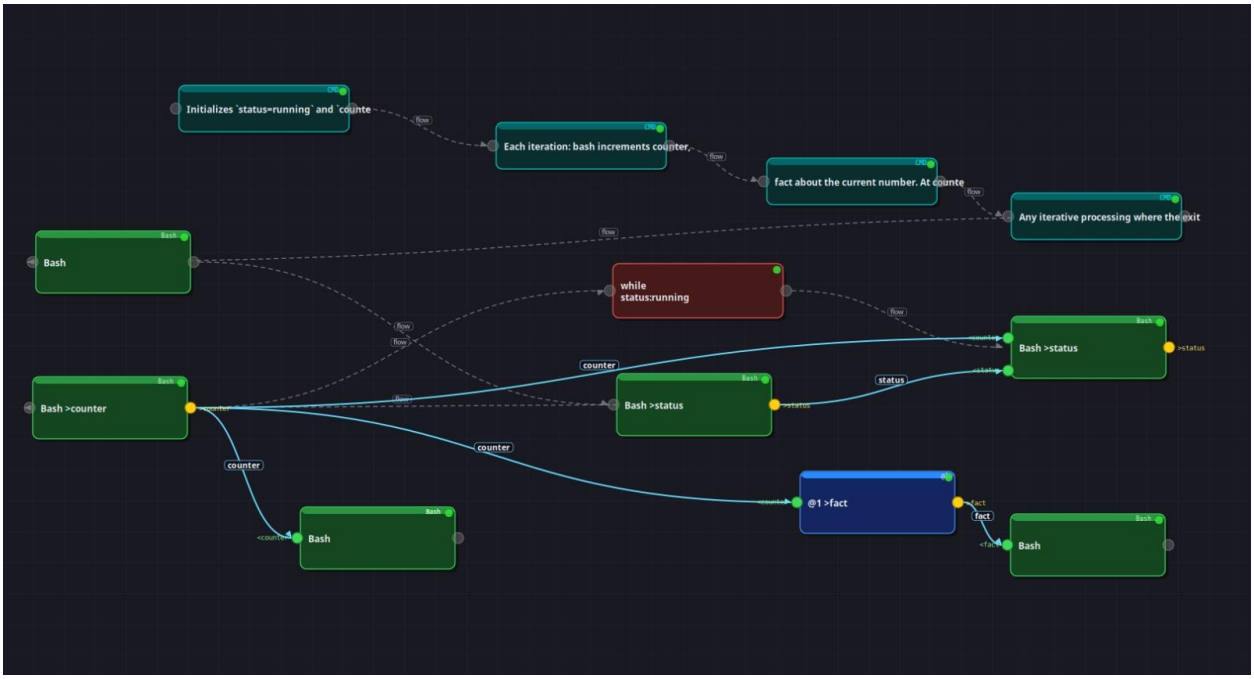
Key concepts: - Multi-`>outvar` bash block must write to `$MSH_VAR_*` directly — stdout not captured. - WHILE reads the **last non-empty line** of the condition variable. - Use `running/done` flag — most reliable exit pattern.

Flow diagram:

```
bash >status="running"
bash >counter="0"
[WHILE status:running]
  bash <counter <status >counter >status
  @1 <counter >fact
  bash <fact
[END_WHILE]
bash <counter
```

Code:

```
bash >status echo "running"

bash >counter echo "0"

bash <counter <status >counter >status val=$(cat "$MSH_VAR_counter")
val=$((val + 1)) echo "$val" > "$MSH_VAR_counter" if [ "$val" -ge 3 ]; then
echo "done" > "$MSH_VAR_status" else   echo "running" > "$MSH_VAR_status" fi
echo "Iteration $val complete"

bash <fact echo "[iter $(cat $MSH_VAR_counter)] $(cat $MSH_VAR_fact)"

bash <counter echo "=== WHILE done. Final counter = $(cat $MSH_VAR_counter)
==="
```



Pattern 14 — FOREACH: LLM Processes Each Item in a List

What it does: Bash creates a list of three UNIX tools with `printf`. FOREACH iterates line by line. LLM describes each tool in one sentence. Bash prints each result in a formatted box.

Use case: Processing any list where each item needs independent LLM analysis.

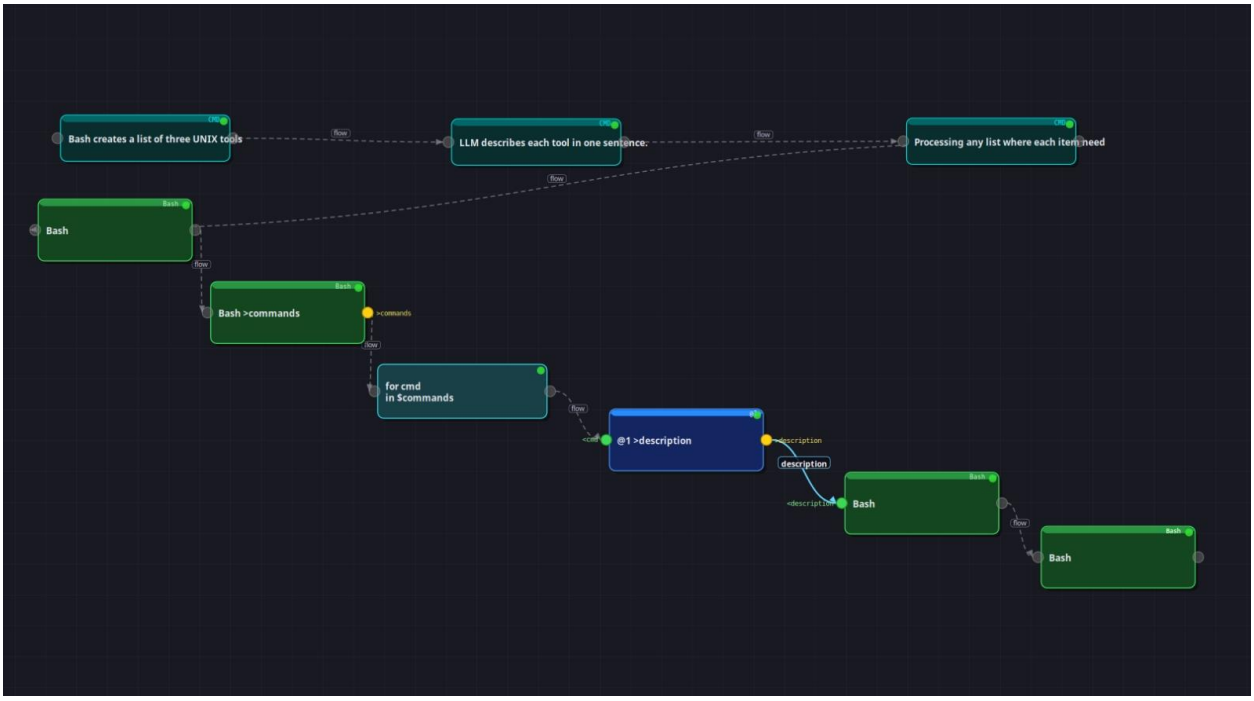
Key concepts: - `printf "a\nb\nc"` for correct line-by-line FOREACH splitting. - Iterator variable set automatically by runtime.

Flow diagram:

```
bash >commands = "awk\nsed\nxargs"
[FOREACH cmd in commands]
  @1 <cmd >description
  bash <description
[END_FOREACH]
bash
```

Code:

```
bash >commands printf "awk\nsed\nxargs"

bash <description echo "━━━ $(cat $MSH_VAR_cmd | tr '[:lower:]' '[:upper:]')
━━━" cat "$MSH_VAR_description" echo ""

echo "=== All commands described ==="
```



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

What it does: Bash initializes a metric string. TRY block uses strict `grep` regex that intentionally fails (`|| exit 1`). CATCH prints hardcoded error message. Safe fallback bash block parses with robust `awk` and prints a bar-chart metrics table.

Use case: Any pipeline where one parsing strategy may fail and a safe fallback must be guaranteed.

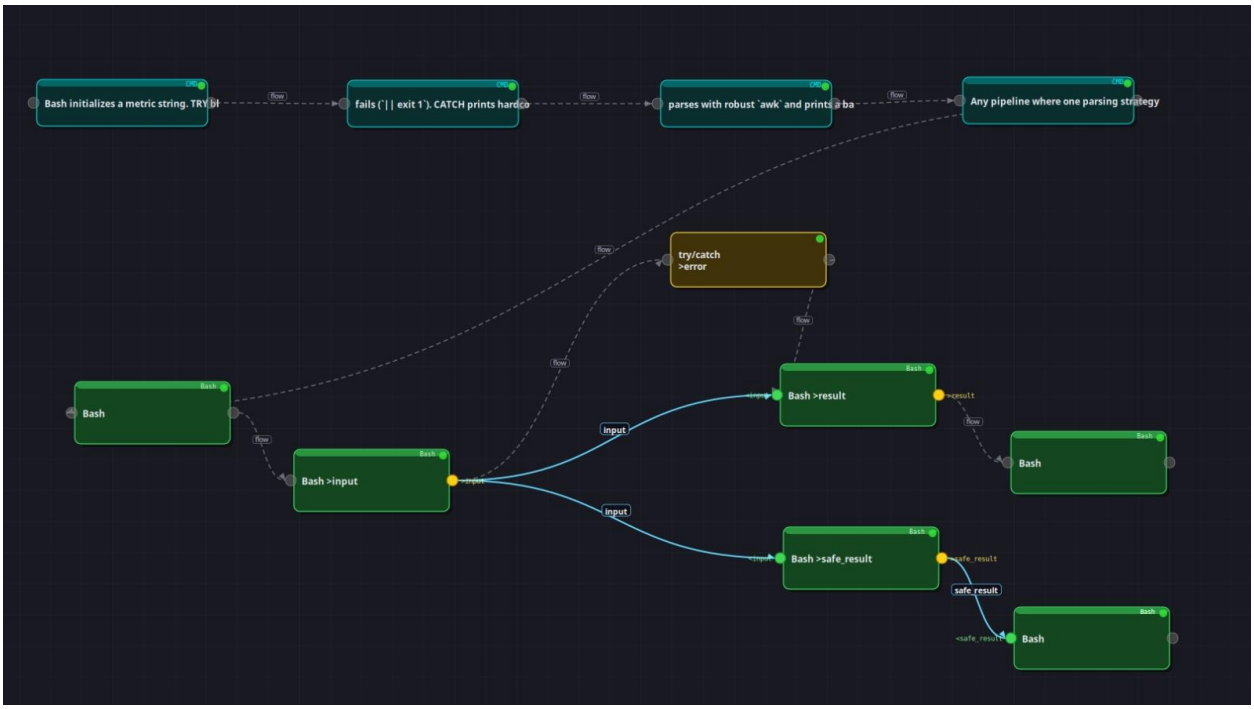
Key concepts: - `|| exit 1` guarantees non-zero exit on any error. - CATCH does **not** use `<errvar` — print `"try_block_failed"` as literal. - Pipeline continues normally after `<!--@end_try-->`.

Flow diagram:

```
bash >input
[TRY]
  bash <input >result    (strict parse → || exit 1)
[CATCH >error]
  bash                   (print "try_block_failed")
[END_TRY]
bash <input >safe_result (robust awk fallback)
bash <safe_result
```

Code:

```
bash >input echo "system:cpu:82 system:mem:91 disk:root:45"

bash <input >result data=$(cat "$MSH_VAR_input") echo "$data" | grep -P
'^\w+=\d+$' || exit 1

echo "=== Caught error: try_block_failed ==="
echo "Strict parser failed. Switching to robust fallback."

bash <input >safe_result data=$(cat "$MSH_VAR_input") echo "=== Metrics
Report ===" echo "$data" | tr ' ' '\n' | awk -F':' '   NF==3 {     pct=$3+0;
bar=""; filled=int(pct/5)     for(i=0;i<filled;i++) bar=bar"▉"
for(i=filled;i<20;i++) bar=bar"▒"     printf "  %-15s %s %3d%%\n", $1"/"$2,
bar, pct   } '

bash <safe_result cat "$MSH_VAR_safe_result"
```



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

What it does: Bash creates a two-line server log dataset. SPLIT marks the logical division into `dataset_1` and `dataset_2`. Two sequential LLMs analyze each part. LLM @1 synthesizes both analyses. Bash prints the unified report.

Use case: Sequential analysis of time-partitioned data (e.g. morning vs. evening shifts) followed by synthesis.

Key concepts: - SPLIT and MERGE are **visual markers** — they execute no code. - Sequential LLM calls guarantee variable availability for the synthesis step.

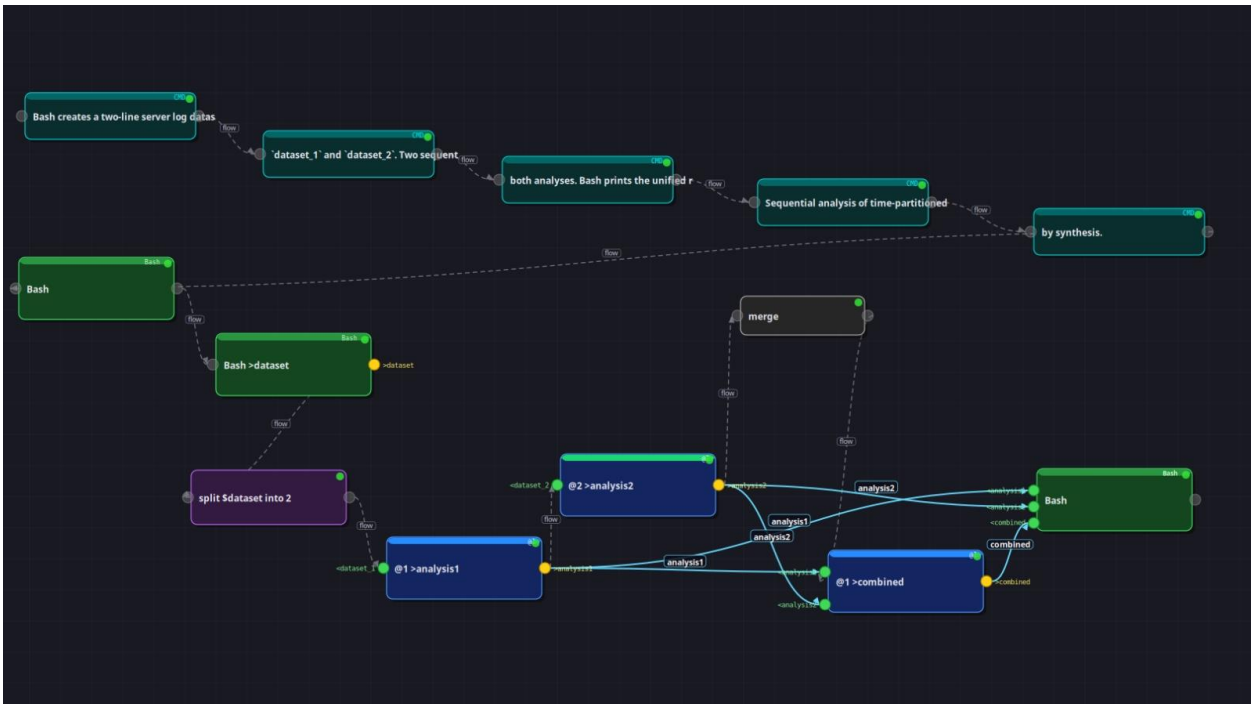
Flow diagram:

```
bash >dataset
[SPLIT dataset into 2]  →  dataset_1, dataset_2
  @1 <dataset_1 >analysis1
  @2 <dataset_2 >analysis2
[MERGE]
  @1 <analysis1 <analysis2 >combined
bash <analysis1 <analysis2 <combined
```

Code:

```
bash >dataset printf "morning: GET /api/users 12400req avg=45ms
errors=12\nevening: GET /api/users 31800req avg=112ms errors=187"
```

```
bash <analysis1 <analysis2 <combined echo "Morning: $(cat
$MSH_VAR_analysis1)" echo "Evening: $(cat $MSH_VAR_analysis2)" echo "Summary:
$(cat $MSH_VAR_combined)"
```



Pattern 17 — CONFIG Node: Parameterized Pipeline

What it does: CONFIG documents `subject`, `style`, `max_words`. Bash blocks set runtime values. LLM @1 explains topic in style. LLM @2 extracts 5 keywords. Bash formats the report with `printf`.

Use case: Reusable pipeline templates — change two bash `echo` lines to switch topic and style.

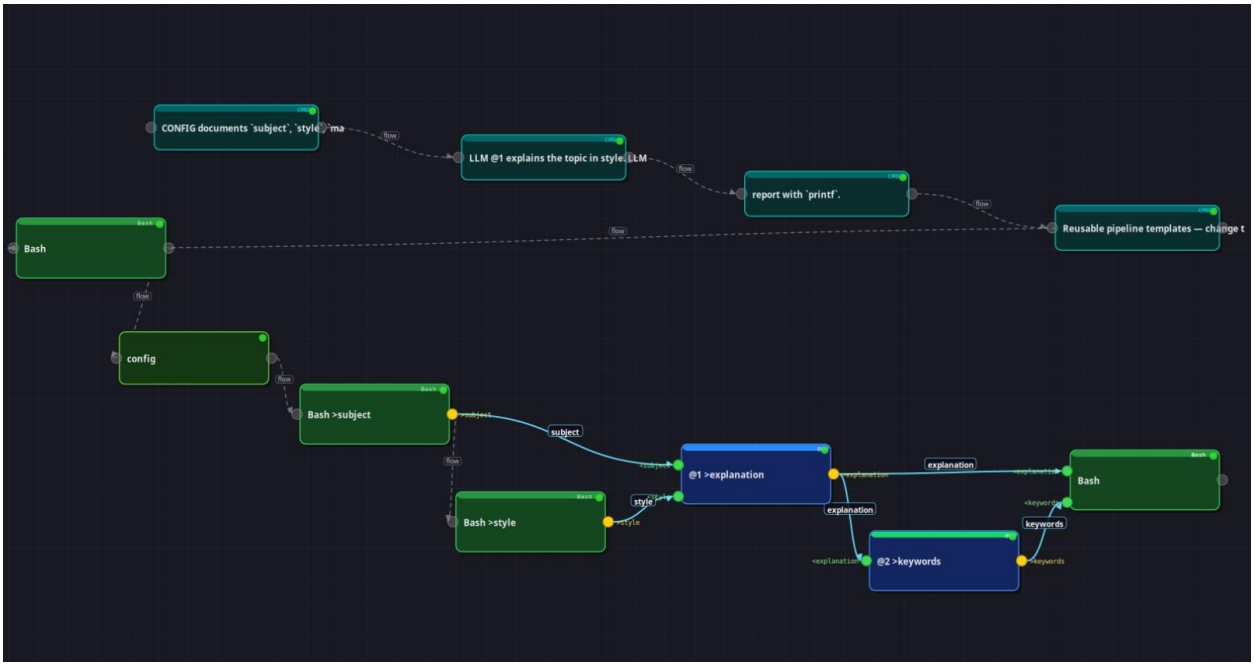
Key concepts: - CONFIG is **documentation only** — always pair with bash `echo` blocks. - Avoid duplicating output with an extra `cat` — mshell captures stdout.

Flow diagram:

```
config  (subject, style, max_words — docs only)
bash >subject >style
@1 <subject <style >explanation
@2 <explanation >keywords
bash <explanation <keywords    (formatted report)
```

Code:

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

bash >subject echo "black holes"

bash >style echo "poetic and accessible"

bash <explanation <keywords echo "Subject  : $(cat $MSH_VAR_subject)" echo
"Style    : $(cat $MSH_VAR_style)" echo "Words    : $(cat
$MSH_VAR_explanation | wc -w)" echo "" echo "Explanation:" && cat
"$MSH_VAR_explanation" echo "" && echo "Keywords: $(cat $MSH_VAR_keywords)"
```



Pattern 18 — FOREACH + Multi-Model: Sequential Batch Processing

What it does: Bash creates a list of three bash concepts. FOREACH iterates. Per item: @1 explains for beginner, @2 gives analogy. Both run sequentially. Bash formats each result in a box.

Use case: Batch processing where each item needs multiple LLM perspectives.

Key concepts: - Sequential LLM calls inside FOREACH are reliable and guaranteed to work. - Each LLM call's output variable is available immediately to subsequent blocks. - `tr '_' ' '` converts underscore-names to readable titles.

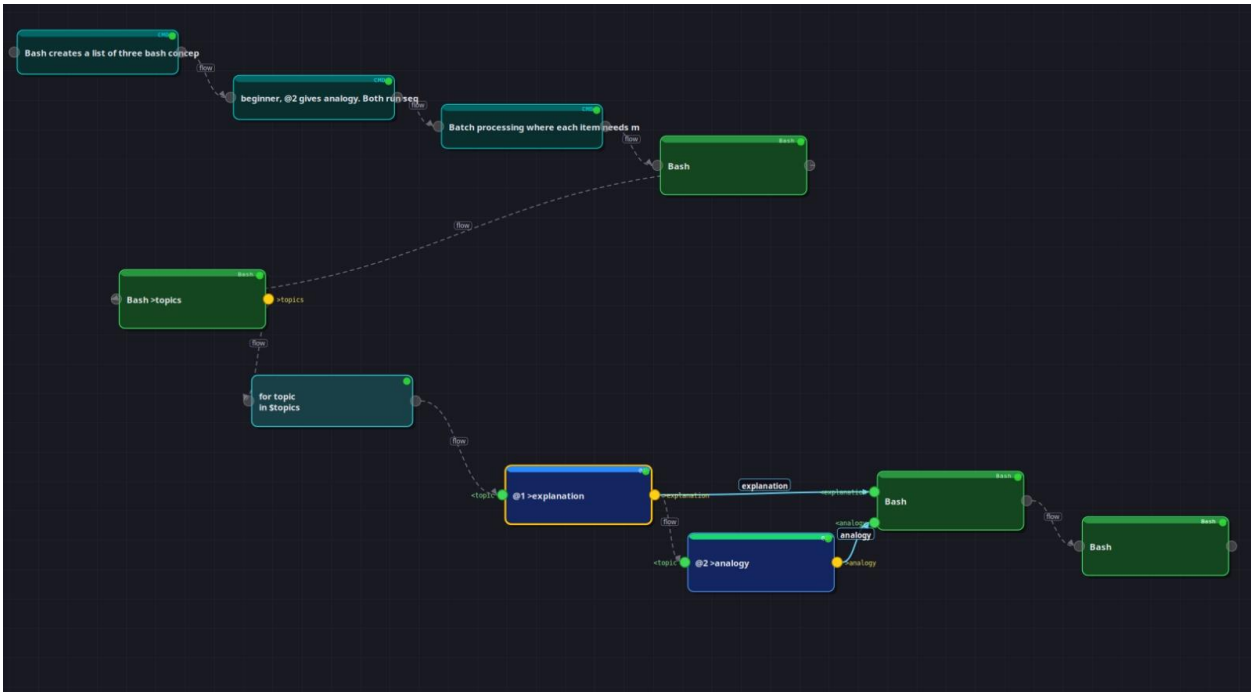
Flow diagram:

```
bash >topics = "process_substitution\nbrace_expansion\nhere_documents"
[FOREACH topic in topics]
  @1 <topic >explanation   (beginner explanation)
```

```
  @2 <topic >analogy       (real-world analogy)
  bash <explanation <analogy
[END_FOREACH]
bash
```

Code:

```
bash >topics printf "process_substitution\nbrace_expansion\nhere_documents"

bash <explanation <analogy echo "┌── $(cat $MSH_VAR_topic | tr '_' ' ')
──┐" echo "  $(cat $MSH_VAR_explanation)" echo "  └→ $(cat $MSH_VAR_analogy)"
echo ""

echo "=== Batch complete ==="
```



Pattern 19 — WHILE Quality Gate: Generate Until Threshold

What it does: WHILE loop generates bash one-liners via LLM @1. LLM @2 scores each on correctness/conciseness 1–10. Loop exits when score ≥ 8. Bash strips whitespace with `tr` before numeric comparison.

Use case: Quality-driven generation — keep trying until a threshold is met.

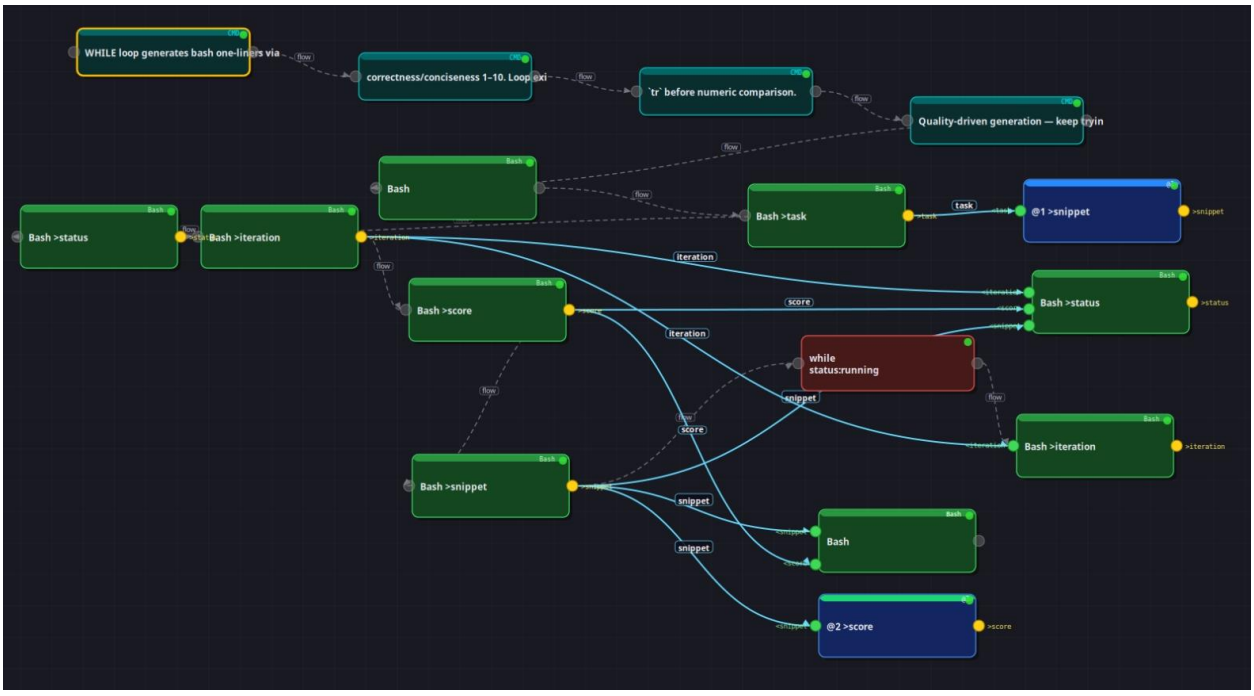
Key concepts: - Scorer must return a bare integer — use `Reply with ONLY the integer.` - `tr -d '[:space:]'` cleans LLM response before `[ "$sc" -ge 8 ]`. - All variables must be initialized before the loop.

Flow diagram:

```
bash >task >status="running" >iteration="0" >score="0" >snippet=""
[WHILE status:running]
  bash <iteration >iteration
  @1 <task >snippet           (generate one-liner)
  @2 <snippet >score          (rate 1-10)
  bash <iteration <score <snippet >status
[END_WHILE]
bash <snippet <score
```

Code:

```
bash >task echo "Write a single bash one-liner that lists the 5 largest files
in the current directory tree, sorted by size descending."

bash >status echo "running"

bash >iteration echo "0"

bash >score echo "0"

bash >snippet echo ""

bash <iteration >iteration val=$(cat "$MSH_VAR_iteration"); echo "$((val +
1))"

bash <iteration <score <snippet >status sc=$(cat "$MSH_VAR_score" | tr -d
'[:space:]') echo "[Iter $(cat $MSH_VAR_iteration)] Score=${sc}: $(cat
$MSH_VAR_snippet)" if [ "$sc" -ge 8 ] 2>/dev/null; then echo "done"; else
echo "running"; fi

bash <snippet <score echo "=== Accepted (score=$(cat $MSH_VAR_score | tr -d
'[:space:]')) ===" cat "$MSH_VAR_snippet"
```



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

What it does: Bash stores a 5-sentence software engineering text. Three bash blocks split into sentence chunks with `awk`. Three sequential LLM @1 calls extract the core principle (3 words each). LLM @2 synthesizes all three into one theme sentence.

Use case: Processing long text in parallel chunks then reducing to a summary.

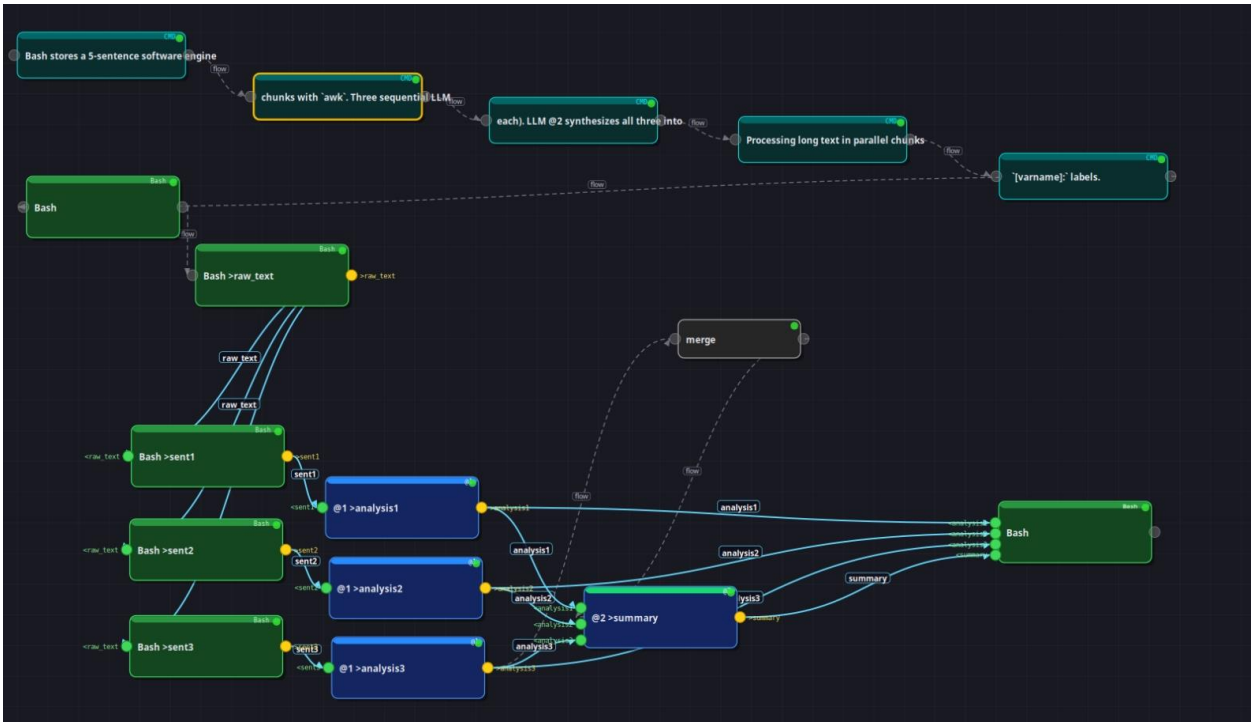
Key concepts: - Map phase: three sequential LLM calls, each reading a different sentence chunk. - Reduce phase: one LLM receives all three analyses via multiple `<invar` with `[varname]:` labels. - SPLIT and MERGE are visual markers only — they execute no code.

Flow diagram:

```
bash >raw_text
bash <raw_text >sent1 / >sent2 / >sent3   (awk split)
  @1 <sent1 >analysis1   MAP: extract principle
  @1 <sent2 >analysis2
  @1 <sent3 >analysis3
[MERGE]
  @2 <analysis1 <analysis2 <analysis3 >summary   (REDUCE)
bash <analysis1 <analysis2 <analysis3 <summary
```

Code:

```
bash >raw_text echo "Clean code is not written for machines but for human
beings who must maintain it. Naming matters more than any other single
decision in software design. Functions should do one thing, do it well, and
do it only. Tests are not overhead – they are the specification of the
system. Refactoring without tests is flying blind in a thunderstorm."

bash <raw_text >sent1 cat "$MSH_VAR_raw_text" | awk -F'. ' '{print $1}'

bash <raw_text >sent2 cat "$MSH_VAR_raw_text" | awk -F'. ' '{print $2". "$3}'

bash <raw_text >sent3 cat "$MSH_VAR_raw_text" | awk -F'. ' '{print $4". "$5}'

bash <analysis1 <analysis2 <analysis3 <summary echo "=== MAP ===" && echo "
$(cat $MSH_VAR_analysis1)" && echo "  $(cat $MSH_VAR_analysis2)" && echo "
$(cat $MSH_VAR_analysis3)" echo "" && echo "=== REDUCE ===" && cat
"$MSH_VAR_summary"
```



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

What it does: LOOP runs up to 3 times. LLM @1 generates a bash script. TRY executes it with `bash "$MSH_VAR_code"`. On success writes `ok`. On failure CATCH writes `fail`. Loop exits when `result == ok`.

Use case: Robust script generation with automatic retry on failure.

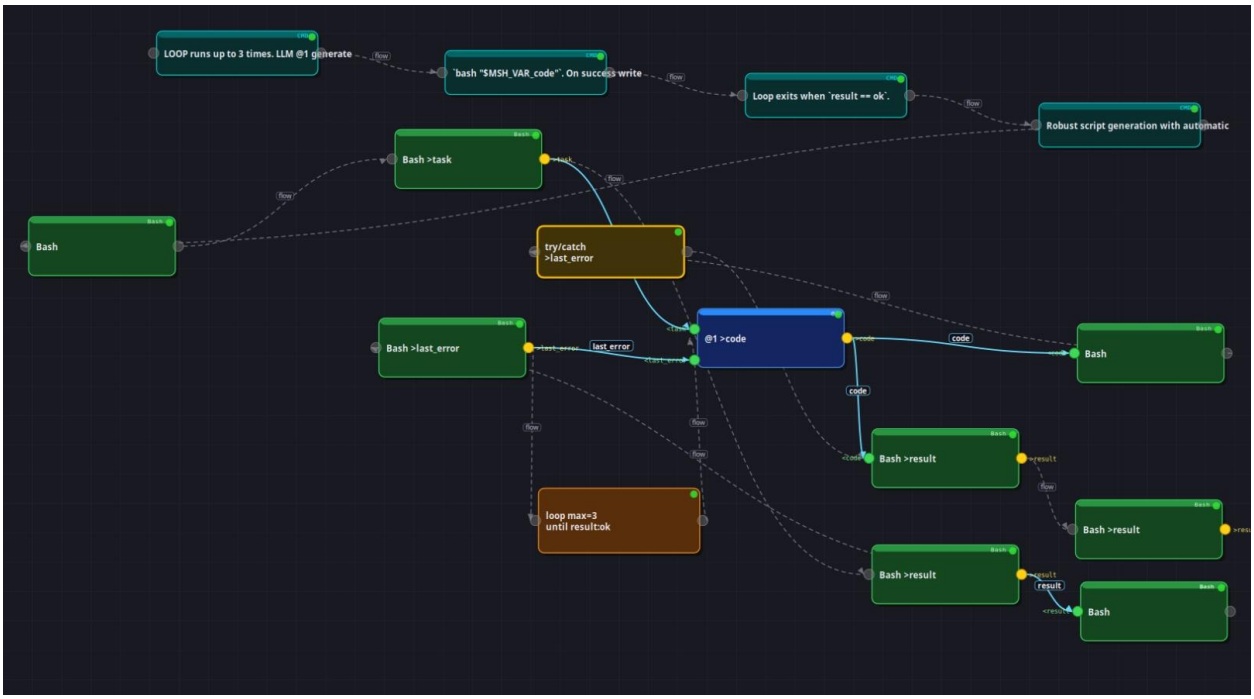
Key concepts: - CATCH does **not** use `<last_error` — writes `fail` directly. - `result` initialized to `"fail"` before loop. - LOOP reads last non-empty line of `result` for `until=` check.

Flow diagram:

```
bash >task >result="fail" >last_error="none"
[LOOP max=3 until=result:ok]
  @1 <task <last_error >code
  bash <code
  [TRY]
    bash <code >result
  [CATCH >last_error]
    bash >result
  [END_TRY]
[END_LOOP]
bash <result
```

Code:

```
bash >task echo "Write a bash script that reads /etc/hosts, counts non-
comment non-empty lines, and prints: 'Active entries: N'."

bash >result echo "fail"

bash >last_error echo "none"

bash <code echo "=== Generated ===" && cat "$MSH_VAR_code" && echo ""

bash <code >result bash "$MSH_VAR_code" echo "ok"

bash >result echo "=== Error: try_block_failed ===" echo "fail"

bash <result echo "=== Final status: $(cat $MSH_VAR_result) ==="
```



Pattern 22 — Multi-Variable Output: Structured Field Extraction

What it does: Bash stores a system alert. LLM @1 responds with strict 3-line format (`SEVERITY:`, `SERVICE:`, `ACTION:`). A bash block with three `>outvar` extracts each field via `grep -oP` and writes directly to `$MSH_VAR_*`. LLM @2 generates a runbook recommendation. Bash prints the incident card.

Use case: Structured field extraction from LLM prose output without Python.

Key concepts: - Multiple `>outvar` on CODE block: must write to `$MSH_VAR_*` directly. - `grep -oP` replaces Python regex for field extraction. - `MSH_VAR_*` environment variables are pre-set by the parser.

Flow diagram:

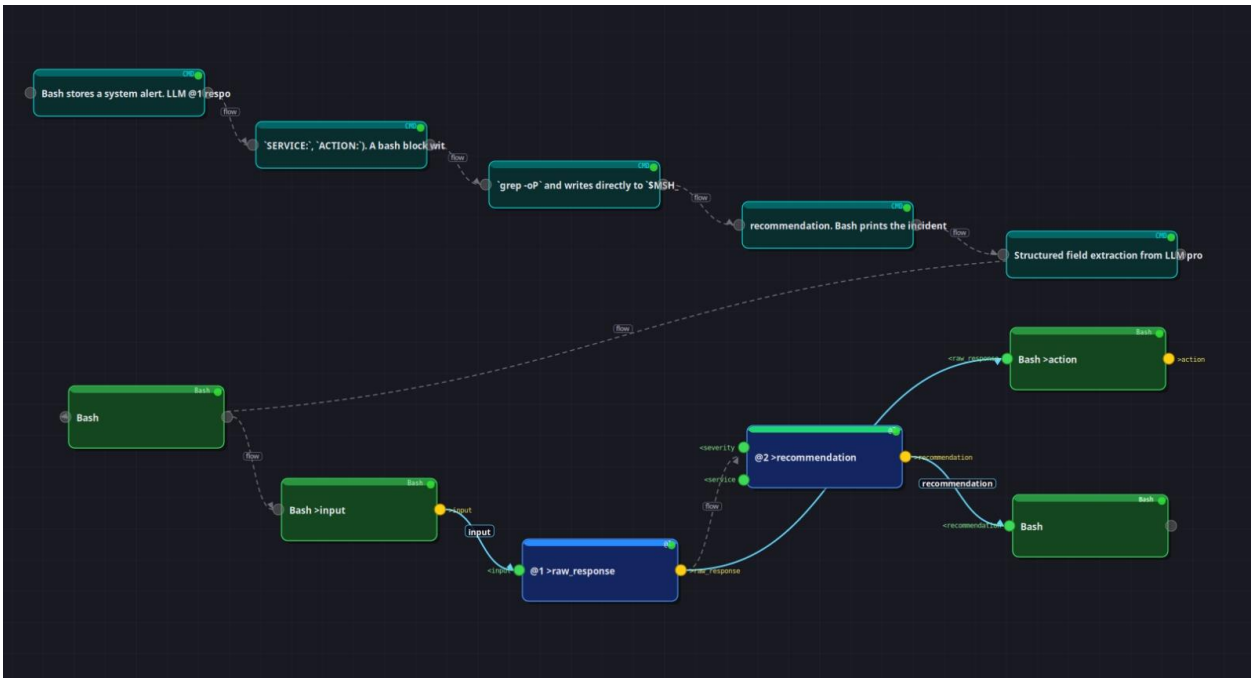
```
bash >input
@1 <input >raw_response    (structured 3-line format)
bash <raw_response >severity >service >action
  (grep fields; write via echo > "$MSH_VAR_*")
@2 <severity <service >recommendation
bash <recommendation        (incident card)
```

Code:

```
bash >input echo "ALERT: Response time on payment-gateway spiked to 4200ms.
Error rate 12.4%. Circuit breaker is OPEN."
```

```
bash <raw_response >severity >service >action text=$(cat
"$MSH_VAR_raw_response") echo "$(echo "$text" | grep -oP
'SEVERITY:\s*\K\S+')" > "$MSH_VAR_severity" echo "$(echo "$text" | grep -oP
'SERVICE:\s*\K\S+')"  > "$MSH_VAR_service" echo "$(echo "$text" | grep -oP
'ACTION:\s*\K.+')"    > "$MSH_VAR_action" echo "Extracted: SEVERITY=$(cat
$MSH_VAR_severity) SERVICE=$(cat $MSH_VAR_service)"
```

```
bash <recommendation echo "SEVERITY : $(cat $MSH_VAR_severity)" echo "SERVICE
: $(cat $MSH_VAR_service)" echo "ACTION   : $(cat $MSH_VAR_action)" echo
"RUNBOOK  : $(cat $MSH_VAR_recommendation)"
```



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

What it does: CONFIG documents parameters. Bash sets runtime values. WHILE loop: @1 generates 3-sentence explanation, @2 scores 1–10 with quality bar chart rendered by bash, loop exits at quality ≥ 7. @3 polishes for publication. Bash prints formatted final result.

Use case: Fully adaptive quality-driven pipeline — reusable template for any topic/audience combination.

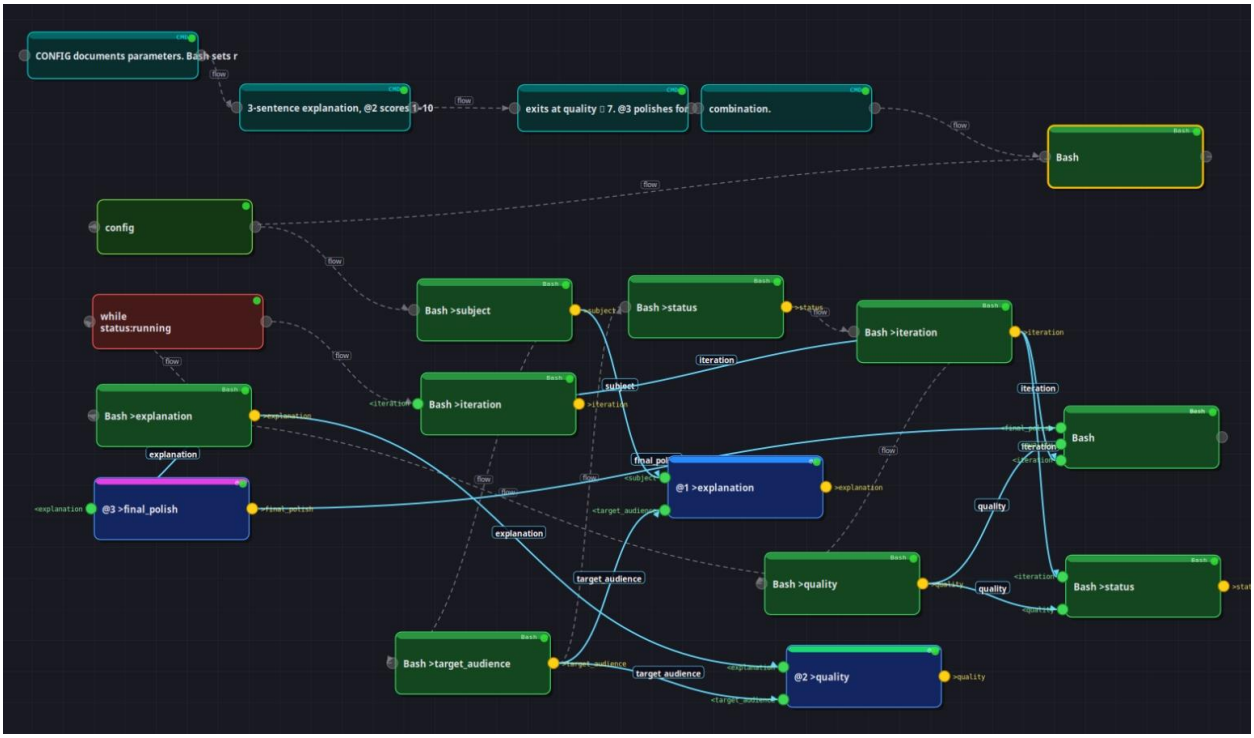
Key concepts: - @2 receives <target_audience explicitly. - Three model roles: generator (@1), scorer (@2), finisher (@3). - Quality bar chart built with bash loop and ▉/▒ characters.

Flow diagram:

```
config + bash >subject >target_audience >status >iteration >quality >explanat
ion
[WHILE status:running]
  bash <iteration >iteration
  @1 <subject <target_audience >explanation
  @2 <explanation <target_audience >quality
  bash <iteration <quality >status
[END_WHILE]
@3 <explanation >final_polish
bash <final_polish <quality <iteration
```

Code: *(see full pattern-23.md)*

```
subject=the CAP theorem in distributed systems
target_audience=junior developer
quality_threshold=7

bash >subject echo "the CAP theorem in distributed systems"

bash >target_audience echo "junior developer"

bash >status echo "running"

bash >iteration echo "0"

bash >quality echo "0"

bash >explanation echo ""

bash <iteration >iteration val=$(cat "$MSH_VAR_iteration"); echo "$((val +
1))"

bash <iteration <quality >status q=$(cat "$MSH_VAR_quality" | tr -d
'[:space:]') bar=""; for i in $(seq 1 10); do [ "$i" -le "$q" ] &&
bar="${bar}▉" || bar="${bar}▒"; done echo "[Iter $(cat $MSH_VAR_iteration)]
${q}/10 [${bar}]" if [ "$q" -ge 7 ] 2>/dev/null; then echo "done"; else echo
"running"; fi

bash <final_polish <quality <iteration echo "Score=$(cat $MSH_VAR_quality |
tr -d '[:space:]')/10  Iters=$(cat $MSH_VAR_iteration)" echo "" && cat
"$MSH_VAR_final_polish"
```



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

What it does: Bash creates a list of log lines (two valid, one broken). FOREACH iterates. Per item: TRY parses with `awk` + strict validation (`|| exit 1`). On success, LLM describes the event. On failure, CATCH prints `[ERR]` as literal. Loop continues regardless.

Use case: Production log batch processing where one malformed record must never halt the pipeline.

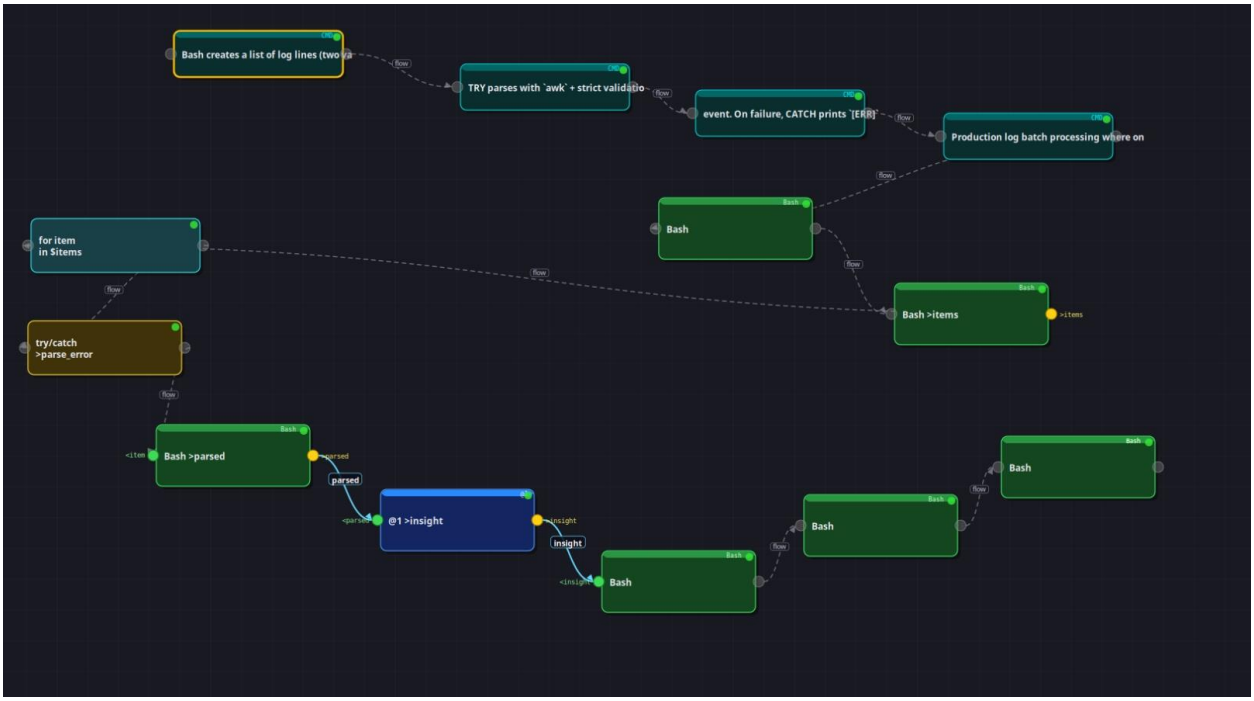
Key concepts: - TRY/CATCH inside FOREACH = **per-item error isolation**. - CATCH does **not** use `<parse_error` — prints literal string. - Do not depend on variables written inside TRY — they may not exist if TRY failed.

Flow diagram:

```
bash >items = "valid_log1\nbroken_log\nvalid_log2"
[FOREACH item in items]
  [TRY]
    bash <item >parsed    (awk strict parse → may exit 1)
    @1 <parsed >insight   (LLM: describe event)
    bash <insight         (print [OK])
  [CATCH >parse_error]
    bash                  (print [ERR] literal)
  [END_TRY]
[END_FOREACH]
bash
```

Code:

```
bash >items printf '2024-01-15T14:32:01Z INFO  api-gateway status=200
duration=45ms\nINFO broken-no-timestamp status=200\n2024-01-15T14:32:05Z
ERROR payment-svc status=500 duration=1240ms'

bash <item >parsed line=$(cat "$MSH_VAR_item") ts=$(echo "$line" | awk
'{print $1}') echo "$ts" | grep -qE '^[0-9]{4}-[0-9]{2}-[0-9]{2}T' || exit 1
svc=$(echo "$line" | awk '{print $3}') [ -z "$svc" ] && exit 1 echo
"timestamp=$ts service=$svc $(echo $line | grep -oE 'status=[0-9]+')"

bash <insight echo "[OK] $(cat $MSH_VAR_insight)"

echo "[ERR] Parse failed: try_block_failed"

echo "=== Batch complete. Errors isolated, pipeline never stopped. ==="
```



Summary Table

#	Pattern	New Nodes	Models	Key Bash Feature
1	Linear Pipeline	—	—	7 bash stages: `printf/awk/rev/tr`
2	LLM in Middle	—	@1	`grep/wc` post-processing of LLM prose
3	Fan-Out	—	@1	`awk` stats + `sort -k2 -rn` from same var
4	Code Gen + Exec	—	@1	`bash "$MSH_VAR_code"`

#	Pattern	New Nodes	Models	Key Bash Feature
5	Two-LLM Review	—	@1, @2	Generate → Review → Improve bash scripts
6	Parallel 3 Models	—	@1–@3	`wc -w` comparison panel
7	Eval-Optimizer Loop	LOOP	@1, @2	`until=` with verdict normalization (`head -1`)
8	Multi-Stage + Multi-Model	—	@1, @2	`awk` replaces C/Python for stats
9	Routing	—	@1	`if=route:VALUE`; 3 bash handlers
10	Full Pipeline	—	@1, @2	Fibonacci in bash; ASCII frame
11	MShell Node	—	@1, @2	`ollama1/2`; bash parses keywords
12	Sequential 3-Model + Synthesis	—	@1–@3	Sequential 3 perspectives; synthesis call
13	WHILE Counter	WHILE	@1	`$MSH_VAR_*` direct write; `running/done`
14	FOREACH List	FOREACH	@1	`printf "a\nb\nc"` list; `tr '_' ' '`
15	TRY/CATCH	TRY/CATCH	—	`|| exit 1`; `awk` bar chart fallback
16	SPLIT + MERGE	SPLIT, MERGE	@1, @2	Sequential log analysis per partition
17	CONFIG Pipeline	CONFIG	@1, @2	`printf` report; CONFIG + bash echo
18	FOREACH + Multi-Model	FOREACH	@1, @2	Per-item sequential; `tr '_' ' '` title
19	WHILE Quality Gate	WHILE	@1, @2	`tr -d '[:space:]'`; bash quality bar
20	Map-Reduce	SPLIT, MERGE	@1, @2	`awk -F'. '` text chunking; sequential map
21	TRY/CATCH + LOOP	TRY/CATCH, LOOP	@1	Retry; `bash "$MSH_VAR_code"`
22	Multi-Var Output	—	@1, @2	`grep -oP`; write via `echo > $MSH_VAR_*`
23	CONFIG+WHILE+3M	CONFIG, WHILE	@1–@3	Bash quality bar with 
24	FOREACH+TRY/CATCH	FOREACH,	@1	`awk` strict log parse; fault

#	Pattern	New Nodes	Models	Key Bash Feature
		TRY/CATCH		isolation

Critical Bash-Specific Rules

1. **Writing multiple output variables:** Use `echo "value" > "$MSH_VAR_varname"` — stdout is **not** captured when multiple `>outvar` are declared.
2. **Reading variables in bash:** Always use `cat "$MSH_VAR_varname"` — never reference the variable name directly.
3. **Score comparison after LLM:** Always strip whitespace before integer comparison:

```
sc=$(cat "$MSH_VAR_score" | tr -d '[:space:]')
[ "$sc" -ge 8 ] 2>/dev/null
```

4. **FOREACH list creation:** Always use `printf "a\nb\nc"` — never `echo "a b c"`.
5. **TRY failure guarantee:** End failing commands with `|| exit 1` to ensure non-zero exit.
6. **CATCH block:** Never use `<errvar` inside CATCH. Print the literal string `"try_block_failed"` directly.
7. **Arithmetic:** Use `$((expr))` for integer math, `bc` for floats/advanced math, `awk` for statistics on datasets.
8. **Field extraction from LLM output:** Use `grep -oP 'LABEL:\s*\K.+'` — this replaces Python `re.search`.
9. **Text splitting for Map-Reduce:** Use `awk -F'. '` for sentence splitting — this replaces Python string methods.
10. **Execute LLM-generated scripts:** Always use `bash "$MSH_VAR_code"` — never source or eval untrusted content in production.
11. **`until=` condition matching in LOOP:** The runtime checks the **last non-empty line** of the variable. When the condition variable comes from an LLM, always normalize it first: `bash head -1 "$MSH_VAR_verdict_raw" | tr -d '[:space:]'` Store the normalized result in a separate variable (e.g. `>verdict`) and use that in `until=verdict:ACCEPTED`.

Appendix I: Code examples for each pattern

Sent to mshell (1929 bytes)

Received from GUI editor:

Pattern 1 Linear Data Pipeline (7 Bash Stages)

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Demonstrates sequential data flow through 7 bash stages. A value produced in Stage 1 is transformed through doubling, hex conversion, string reversal, statistics computation, uppercasing, and finally ASCII-frame formatting. Each stage reads the output of the previous block and passes its own output to the next.

Use case

Multi-stage ETL pipelines where each bash block handles a specific transformation \u2014 entirely without external language runtimes.

Key concept

Variables flow strictly **top-to-bottom**. Each producer must appear before its consumers in the document.

Flow diagram

```
bash >raw        (generate integer 42)
bash <raw >doubled       (×2, label)
bash <doubled >hexval    (printf '%x')
bash <hexval >reversed   (rev)
bash <reversed >stats    (wc -c / wc -w)
bash <stats >upper       (tr to uppercase)
bash <upper              (ASCII-frame report)
```

Code

```bash >raw
echo "42"
```

```bash <raw >doubled
val=$(cat "$MSH_VAR_raw")
result=$((val * 2))
echo "value=${val} doubled=${result}"
```

```bash <doubled >hexval
text=$(cat "$MSH_VAR_doubled")
num=$(echo "$text" | grep -oP 'doubled=\K[0-9]+')
hex=$(printf '%x' "$num")
echo "hex_of_doubled=${hex}"
```

```bash <hexval >reversed
line=$(cat "$MSH_VAR_hexval")
hex=$(echo "$line" | grep -oP 'hex_of_doubled=\K\S+')
rev_hex=$(echo "$hex" | rev)
echo "reversed_hex=${rev_hex}"
```

```bash <reversed >stats
line=$(cat "$MSH_VAR_reversed")
chars=$(echo "$line" | wc -c)
words=$(echo "$line" | wc -w)
echo "chars=${chars} words=${words} data=${line}"
```

```bash <stats >upper
cat "$MSH_VAR_stats" | tr '[:lower:]' '[:upper:]'
```

```bash <upper
line=$(cat "$MSH_VAR_upper")
border=$(printf '%0.s=' $(seq 1 ${#line}))
echo "$border"
echo "$line"
echo "$border"
echo "=== Pipeline complete: 7 bash stages ==="
```

42
value=42 doubled=84
hex_of_doubled=54
reversed_hex=45
chars=16 words=1 data=reversed_hex=45
CHARS=16 WORDS=1 DATA=REVERSED_HEX=45
==========
CHARS=16 WORDS=1 DATA=REVERSED_HEX=45
==========
=== Pipeline complete: 7 bash stages ===
/home/igor >

/home/igor > Sent to mshell (1861 bytes)
Received from GUI editor:

Pattern 2 LLM in the Middle

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

A bash block generates structured server log data. LLM @1 receives the log and produces a one-paragraph incident analysis. A second bash block post-processes the LLM response: it counts sentences, extracts numbers, and prints a formatted digest using `grep`, `awk`, `tr`, and `wc`.

Use case

Automated data analysis pipelines where you want a natural language interpretation of structured data, followed by further extraction of metadata from the prose response.

Key concept

LLM directive `<!--@1 <data >result ... -->` takes input from a variable, sends it as part of the prompt, and stores the model response in an output variable accessible to subsequent blocks.

Flow diagram

```
bash >log_data        (structured server log summary)
@1 <log_data >analysis  (LLM: incident analysis paragraph)
bash <analysis        (extract stats, print digest)
```

Code

```bash >log_data
cat <<'EOF'
Server log summary (last 1h):
200 OK       : 8420 requests  avg=42ms
301 Redirect  : 312  requests  avg=5ms
400 Bad Req   : 87   requests  avg=12ms
```

404 Not Found : 203 requests avg=8ms

500 Error   : 34   requests avg=890ms

503 Unavail : 11   requests avg=2100ms

EOF

```

<!--@1 <log_data >analysis

The input is a server log summary. Write one paragraph identifying the main concern, which status codes are alarming, and what immediate action you recommend.

-->

```bash <analysis

text=$(cat "$MSH_VAR_analysis")

sentences=$(echo "$text" | grep -oE '[^.!?]+[.!?]' | wc -l)

numbers=$(echo "$text" | grep -oE '[0-9]+' | tr '\n' ',' | sed 's/,$//')

echo "=== LLM Incident Analysis ==="

echo "$text"

echo ""

echo "--- Digest ---"

echo "Sentences : $sentences"

echo "Numbers mentioned : $numbers"

echo "Word count : $(echo "$text" | wc -w)"

```

Server log summary (last 1h):

200 OK   : 8420 requests avg=42ms

301 Redirect : 312 requests avg=5ms

400 Bad Req   : 87   requests avg=12ms

404 Not Found : 203 requests avg=8ms
```

500 Error   : 34   requests avg=890ms

503 Unavail   : 11   requests avg=2100ms

The primary concern in this log summary is the presence of critical server errors, particularly the 500 Internal Server Errors and 503 Service Unavailable responses, which indicate serious backend issues. While the 500 errors represent only 0.38% of total requests (34 out of 9,067), their extremely high average response time of 890ms suggests significant processing problems, and the 11 instances of 503 errors with an alarming 2,100ms average response time indicate the server is occasionally completely overwhelmed or experiencing resource exhaustion. The immediate action should be to investigate the root cause of these server-side failures by examining detailed error logs, checking system resources (CPU, memory, disk space), monitoring database performance, and reviewing any recent deployments or configuration changes that might have introduced instability - these critical errors need to be resolved quickly to prevent service degradation and potential downtime.

=== LLM Incident Analysis ===

The primary concern in this log summary is the presence of critical server errors, particularly the 500 Internal Server Errors and 503 Service Unavailable responses, which indicate serious backend issues. While the 500 errors represent only 0.38% of total requests (34 out of 9,067), their extremely high average response time of 890ms suggests significant processing problems, and the 11 instances of 503 errors with an alarming 2,100ms average response time indicate the server is occasionally completely overwhelmed or experiencing resource exhaustion. The immediate action should be to investigate the root cause of these server-side failures by examining detailed error logs, checking system resources (CPU, memory, disk space), monitoring database performance, and reviewing any recent deployments or configuration changes that might have introduced instability - these critical errors need to be resolved quickly to prevent service degradation and potential downtime.

--- Digest ---

Sentences : 4

Numbers mentioned : 500,503,500,0,38,34,9,067,890,11,503,2,100

Word count : 139

/home/igor >

/home/igor > Sent to mshell (1764 bytes)

Received from GUI editor:

Pattern 3 Fan-Out: One Variable - Many Consumers

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

A single CSV dataset of city temperatures is produced once and consumed independently by three downstream blocks: bash computes statistics with `awk`, bash sorts cities by temperature, and LLM @1 writes a weather narrative.

Use case

Parallel analysis of the same dataset from different perspectives without duplicating data.

Key concept

Multiple blocks can reference the same `<varname`. All consumers read the same unchanged file.

Flow diagram

```
bash >cities           (CSV: city,temp_celsius)
bash <cities >stats     (awk: min/max/avg)
bash <cities >sorted     (sort descending)
@1  <cities >narrative   (LLM: weather sentence)
bash <stats <sorted <narrative  (unified report)
```

Code

```bash >cities
printf "London,12\nParis,18\nMoscow,-3\nDubai,38\nTokyo,22\nNewYork,9\nSydney,25\n"
```

```bash <cities >stats
data=$(cat "$MSH_VAR_cities")
result=$(echo "$data" | awk -F',' '
  BEGIN { min=9999; max=-9999; sum=0; n=0 }
```

```
  { t=$2+0; if(t<min) min=t; if(t>max) max=t; sum+=t; n++ }
  END { printf "count=%d min=%d max=%d avg=%.1f", n, min, max, sum/n }
')
echo "$result"
```

```bash <cities >sorted
echo "=== Cities by temperature (desc) ==="
cat "$MSH_VAR_cities" | sort -t',' -k2 -rn | \
  awk -F',' '{ printf "  %-12s %3d°C\n", $1, $2 }'
```

<!--@1 <cities >narrative
The input is a CSV list of city names and temperatures in Celsius.
Write exactly one sentence describing the global temperature spread shown.
-->

```bash <stats <sorted <narrative
echo "=== Statistics ===" && cat "$MSH_VAR_stats"
echo "" && cat "$MSH_VAR_sorted"
echo "" && echo "=== AI Narrative ===" && cat "$MSH_VAR_narrative"
```

London,12
Paris,18
Moscow,-3
Dubai,38
Tokyo,22
NewYork,9
Sydney,25

count=7 min=-3 max=38 avg=17.3

=== Cities by temperature (desc) ===

Dubai 38°C

Sydney 25°C

Tokyo 22°C

Paris 18°C

London 12°C

NewYork 9°C

Moscow -3°C

The temperature data shows a global spread ranging from Moscow's freezing -3°C to Dubai's hot 38°C, with most cities experiencing mild to warm temperatures between 9°C and 25°C.

=== Statistics ===

count=7 min=-3 max=38 avg=17.3

=== Cities by temperature (desc) ===

Dubai 38°C

Sydney 25°C

Tokyo 22°C

Paris 18°C

London 12°C

NewYork 9°C

Moscow -3°C

=== AI Narrative ===

The temperature data shows a global spread ranging from Moscow's freezing -3°C to Dubai's hot 38°C, with most cities experiencing mild to warm temperatures between 9°C and 25°C./home/igor >

/home/igor > Sent to mshell (1194 bytes)

Received from GUI editor:

Pattern 4 LLM Code Generation - Execute via Variable

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

A bash block stores a shell task description. LLM @1 generates a **pure bash script**

that solves it. A second bash block inspects and executes the script via

`bash "$MSH_VAR_code"`.

Use case

Automated bash script generation workflows \u2014 no copy-paste needed.

Key concept

`$MSH_VAR_code` contains the file path to the generated script.

`bash "$MSH_VAR_code"` executes it directly.

Flow diagram

```

bash >task

@1 <task >code    (LLM: generate pure bash script, no fences)

bash <code       (inspect + execute)

```

Code

```bash >task

echo "Write a bash script that takes the sentence 'the cat sat on the mat and the cat sat' and prints each word that appears more than once along with its count, sorted by count descending."

```

<!--@1 <task >code

The input describes a bash scripting task.

Write ONLY a pure bash script (no Python, no other interpreters).

No markdown fences. No explanation. Just the script.

-->

```bash <code
echo "=== Generated bash script ==="
cat "$MSH_VAR_code"
echo ""
echo "=== Executing ==="
bash "$MSH_VAR_code"
```

Write a bash script that takes the sentence 'the cat sat on the mat and the cat sat' and prints each word that appears more than once along with its count, sorted by count descending.

```
#!/bin/bash
sentence="the cat sat on the mat and the cat sat"
declare -A word_count
for word in $sentence; do
  ((word_count[$word]++))
done
for word in "${!word_count[@]}"; do
  if [ ${word_count[$word]} -gt 1 ]; then
    echo "$word ${word_count[$word]}"
  fi
done | sort -k2 -nr
=== Generated bash script ===
#!/bin/bash
sentence="the cat sat on the mat and the cat sat"
declare -A word_count
```

```
for word in $sentence; do
   ((word_count[$word]++))
done
for word in "${!word_count[@]}"; do
   if [ ${word_count[$word]} -gt 1 ]; then
      echo "$word ${word_count[$word]}"
   fi
done | sort -k2 -nr
=== Executing ===
the 3
sat 2
cat 2
/home/igor >
------------------------------------------------------------
/home/igor > Sent to mshell (1911 bytes)
Received from GUI editor:
----------------------------------------
# Pattern 5 Two-LLM Review Chain
> **Edition:** Bash-Only · Art2Dec SoftLab, 2026
## What it does
LLM @1 generates a bash script. LLM @2 reviews it. LLM @1 receives both the script and
the review, produces an improved version. A final bash block executes it.
## Use case
Automated bash script quality improvement \u2014 generate, review, and refine in a single
pipeline.
## Key concept
The second LLM call passes both `<code` and `<review` as inputs. Multiple input variables
```

are concatenated with `[varname]:` labels automatically.

Flow diagram

```

bash >task

@1 <task >code          (generate bash script)

bash <code              (print)

@2 <code >review        (review for correctness)

bash <review            (print)

@1 <code <review >improved   (improve)

bash <improved          (print + execute)

```

Code

```bash >task

echo "Write a bash function is_valid_ipv4 that takes one argument and returns 0 (true) if it is a valid IPv4 address, 1 (false) otherwise. Test it on: 192.168.1.1, 256.0.0.1, 10.0.0, 0.0.0.0, and 999.999.999.999"

```

<!--@1 <task >code

Write only a pure bash script, no markdown fences, no explanation.

The script must define the function and run the test cases described.

-->

```bash <code

echo "=== Model 1 generated ==="

cat "$MSH_VAR_code"

```

<!--@2 <code >review

Review this bash script strictly.

Check: correct validation for all 4 octets 0-255, edge cases (leading zeros, empty string, too many octets).

Reply in 2 sentences max.

-->

```bash <review
echo "=== Model 2 review ==="
cat "$MSH_VAR_review"
```

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

The first input is a bash script. The second input is a code review.

Improve the script based on the review. Return ONLY the improved pure bash script, no fences.

-->

```bash <improved
echo "=== Final improved script ==="
cat "$MSH_VAR_improved"
echo "" && echo "=== Executing ==="
bash "$MSH_VAR_improved"
```

Write a bash function is_valid_ipv4 that takes one argument and returns 0 (true) if it is a valid IPv4 address, 1 (false) otherwise. Test it on: 192.168.1.1, 256.0.0.1, 10.0.0, 0.0.0.0, and 999.999.999.999

```
#!/bin/bash
is_valid_ipv4() {
  local ip="$1"
  if [[ ! "$ip" =~ ^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
    return 1
```

```
    fi
    IFS='.' read -ra octets <<< "$ip"

    if [ ${#octets[@]} -ne 4 ]; then
        return 1
    fi
    for octet in "${octets[@]}"; do
        if [ "$octet" -lt 0 ] || [ "$octet" -gt 255 ]; then
            return 1
        fi
        if [ ${#octet} -gt 1 ] && [[ "$octet" =~ ^0 ]]; then
            return 1
        fi
    done
    return 0
}
test_cases=("192.168.1.1" "256.0.0.1" "10.0.0" "0.0.0.0" "999.999.999.999")
for ip in "${test_cases[@]}"; do
    if is_valid_ipv4 "$ip"; then
        echo "$ip: valid"
    else
        echo "$ip: invalid"
    fi
done
=== Model 1 generated ===
#!/bin/bash
is_valid_ipv4() {
```

```
    local ip="$1"
    if [[ ! "$ip" =~ ^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
        return 1
    fi
    IFS='.' read -ra octets <<< "$ip"
    if [ ${#octets[@]} -ne 4 ]; then
        return 1
    fi
    for octet in "${octets[@]}"; do
        if [ "$octet" -lt 0 ] || [ "$octet" -gt 255 ]; then
            return 1
        fi
        if [ ${#octet} -gt 1 ] && [[ "$octet" =~ ^0 ]]; then
            return 1
        fi
    done
    return 0
}
test_cases=("192.168.1.1" "256.0.0.1" "10.0.0" "0.0.0.0" "999.999.999.999")
for ip in "${test_cases[@]}"; do
    if is_valid_ipv4 "$ip"; then
        echo "$ip: valid"
    else
        echo "$ip: invalid"
    fi
```

done The script correctly validates IPv4 addresses (0\u2013255 per octet, no leading zeros except for single 0, rejects empty string, too few/many octets) for typical bash environments, but relies on bash\u2019s `[[ =~ ]]` and arithmetic evaluation, so it\u2019s

not strictly POSIX\u2011sh compatible. One subtle edge case: numbers like `08` or `09` are handled correctly as invalid due to the leading-zero check, and extremely large numeric octets could trigger integer overflow in the `[ "$octet" -gt 255 ]` test on some systems, though that\u2019s unlikely in practice.

=== Model 2 review ===

The script correctly validates IPv4 addresses (0\u2013255 per octet, no leading zeros except for single 0, rejects empty string, too few/many octets) for typical bash environments, but relies on bash\u2019s `[[ =~ ]]` and arithmetic evaluation, so it\u2019s not strictly POSIX\u2011sh compatible. One subtle edge case: numbers like `08` or `09` are handled correctly as invalid due to the leading-zero check, and extremely large numeric octets could trigger integer overflow in the `[ "$octet" -gt 255 ]` test on some systems, though that\u2019s unlikely in practice.

```
#!/bin/bash

is_valid_ipv4() {
  local ip="$1"
  if [[ ! "$ip" =~ ^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
    return 1
  fi
  IFS='.' read -ra octets <<< "$ip"
  if [ ${#octets[@]} -ne 4 ]; then
    return 1
  fi
  for octet in "${octets[@]}"; do
    if [ ${#octet} -gt 3 ]; then
      return 1
    fi
    if [ ${#octet} -gt 1 ] && [[ "$octet" =~ ^0 ]]; then
      return 1
    fi
    if [ "$octet" -gt 255 ] 2>/dev/null; then
      return 1
```

```
        fi
        if ! [ "$octet" -ge 0 ] 2>/dev/null; then
            return 1
        fi
    done
    return 0
}
test_cases=("192.168.1.1" "256.0.0.1" "10.0.0" "0.0.0.0" "999.999.999.999")
for ip in "${test_cases[@]}"; do
    if is_valid_ipv4 "$ip"; then
        echo "$ip: valid"
    else
        echo "$ip: invalid"
    fi
done
=== Final improved script ===
#!/bin/bash
is_valid_ipv4() {
    local ip="$1"
    if [[ ! "$ip" =~ ^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
        return 1
    fi
    IFS='.' read -ra octets <<< "$ip"
    if [ ${#octets[@]} -ne 4 ]; then
        return 1
    fi
    for octet in "${octets[@]}"; do
```

```
        if [ ${#octet} -gt 3 ]; then
            return 1
        fi
        if [ ${#octet} -gt 1 ] && [[ "$octet" =~ ^0 ]]; then
            return 1
        fi
        if [ "$octet" -gt 255 ] 2>/dev/null; then
            return 1
        fi
        if ! [ "$octet" -ge 0 ] 2>/dev/null; then
            return 1
        fi
    done
    return 0
}
test_cases=("192.168.1.1" "256.0.0.1" "10.0.0" "0.0.0.0" "999.999.999.999")
for ip in "${test_cases[@]}"; do
    if is_valid_ipv4 "$ip"; then
        echo "$ip: valid"
    else
        echo "$ip: invalid"
    fi
done
=== Executing ===
192.168.1.1: valid
256.0.0.1: invalid
10.0.0: invalid
```

0.0.0.0: valid

999.999.999.999: invalid

/home/igor >

/home/igor > Sent to mshell (1412 bytes)

Received from GUI editor:

Pattern 6 Parallel 3-Model Query

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

The same system design question is sent to all three LLM models independently. A final bash block collects all three responses and prints a comparison panel with word counts.

Use case

Comparing answers across models, collecting ensemble responses, or generating diverse perspectives on the same topic.

Key concept

Three LLM directive blocks with `@1`, `@2`, `@3` all read the same `<question` and write to separate `>ans1`, `>ans2`, `>ans3`.

Flow diagram

```
bash >question
@1 <question >ans1
@2 <question >ans2
@3 <question >ans3
bash <ans1 <ans2 <ans3   (comparison panel)
```

Code

```bash >question

echo "You are designing a URL shortener service expecting 100M requests/day. In one sentence, name the single most critical architectural decision and why."

```

<!--@1 <question >ans1

Answer the question in exactly one sentence.

-->

<!--@2 <question >ans2

Answer the question in exactly one sentence.

-->

<!--@3 <question >ans3

Answer the question in exactly one sentence.

-->

```bash <ans1 <ans2 <ans3

w1=$(cat "$MSH_VAR_ans1" | wc -w)

w2=$(cat "$MSH_VAR_ans2" | wc -w)

w3=$(cat "$MSH_VAR_ans3" | wc -w)

echo "\u25b6 Model @1 (${w1}w): $(cat $MSH_VAR_ans1)"

echo ""

echo "\u25b6 Model @2 (${w2}w): $(cat $MSH_VAR_ans2)"

echo ""

echo "\u25b6 Model @3 (${w3}w): $(cat $MSH_VAR_ans3)"

```

You are designing a URL shortener service expecting 100M requests/day. In one sentence, name the single most critical architectural decision and why.

The most critical architectural decision is implementing an efficient distributed caching layer (like Redis) because with 100M requests/day, the read-heavy workload of URL lookups will overwhelm any database without fast, scalable caching to handle the 1,157 requests per second average load.

The single most critical architectural decision is choosing a horizontally scalable, partitioned datastore and ID-generation scheme (e.g., sharded database with globally unique, non-colliding short IDs) because it directly determines the system\u2019s ability to handle high write throughput, ensure uniqueness, and remain available as traffic grows.

The single most critical architectural decision is implementing a distributed, horizontally scalable key-value store for URL mappings to handle the high read throughput (100M requests/day) with low latency and ensure fault tolerance without becoming a bottleneck.

\u25b6 Model @1 (41w): The most critical architectural decision is implementing an efficient distributed caching layer (like Redis) because with 100M requests/day, the read-heavy workload of URL lookups will overwhelm any database without fast, scalable caching to handle the 1,157 requests per second average load.

\u25b6 Model @2 (45w): The single most critical architectural decision is choosing a horizontally scalable, partitioned datastore and ID-generation scheme (e.g., sharded database with globally unique, non-colliding short IDs) because it directly determines the system\u2019s ability to handle high write throughput, ensure uniqueness, and remain available as traffic grows.

\u25b6 Model @3 (36w): The single most critical architectural decision is implementing a distributed, horizontally scalable key-value store for URL mappings to handle the high read throughput (100M requests/day) with low latency and ensure fault tolerance without becoming a bottleneck.

/home/igor > Sent to mshell (2182 bytes)

Received from GUI editor:

Pattern 7 Evaluator-Optimizer Loop

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

LLM @1 generates a bash script. LLM @2 returns `ACCEPTED` or `REJECTED`. A bash

normalization block strips the verdict to its first line so the `until=` condition

matches reliably. The loop repeats until acceptance or max iterations. Final script is executed.

Use case

Automated iterative refinement with quality gates.

Key concept

`<!--@loop max=N until=verdict:ACCEPTED-->` wraps the generator-evaluator pair. Exits early on acceptance or after N iterations.

Critical rule

`until=verdict:ACCEPTED` checks the **last non-empty line** of the variable. LLM @2 must return `ACCEPTED` or `REJECTED` as a **single word, nothing else**. A bash normalization block after `@2` extracts only the first line to ensure the condition variable is clean.

Flow diagram

```
bash >task
[LOOP max=3 until=verdict:ACCEPTED]
 @1 <task >code          (generate bash script)
 bash <code              (print)
 @2 <code >verdict_raw     (ACCEPTED or REJECTED, one word only)
 bash <verdict_raw >verdict (normalize: head -1, strip whitespace)
 bash <verdict           (print)
[END_LOOP]
bash <code               (execute accepted script)
```

Code

```bash >task
```

````
echo "Write a bash function seconds_to_human that converts seconds to '2h 5m 30s'. Omit zero units. Test with: 0, 59, 3661, 86400, 90061."

```

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

<!--@1 <task >code

Write only a pure bash script. No markdown fences. No explanation.

Define the function and call it with the test values listed.

-->

```bash <code

echo "=== Generated script ==="

cat "$MSH_VAR_code"

```

<!--@2 <code >verdict_raw

Review this bash script. Check arithmetic correctness for h/m/s extraction and zero-unit omission.

Reply with ONLY one word: ACCEPTED or REJECTED. No explanation. No punctuation.

-->

```bash <verdict_raw >verdict

head -1 "$MSH_VAR_verdict_raw" | tr -d '[:space:]'

```

```bash <verdict

echo "=== Verdict ===" && cat "$MSH_VAR_verdict" && echo ""

```

<!--@end_loop-->

```bash <code

echo "=== Final accepted script \u2014 executing ==="

bash "$MSH_VAR_code"

```
````
````

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

Write a bash function seconds_to_human that converts seconds to '2h 5m 30s'. Omit zero units. Test with: 0, 59, 3661, 86400, 90061.

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

```
#!/bin/bash

seconds_to_human() {
  local total_seconds="$1"
  local result=""
  if [ "$total_seconds" -eq 0 ]; then
    echo "0s"
    return
  fi
  local hours=$((total_seconds / 3600))
  local minutes=$(((total_seconds % 3600) / 60))
  local seconds=$((total_seconds % 60))
  if [ "$hours" -gt 0 ]; then
    result="${result}${hours}h "
  fi
  if [ "$minutes" -gt 0 ]; then
    result="${result}${minutes}m "
  fi
  if [ "$seconds" -gt 0 ]; then
    result="${result}${seconds}s"
  fi
  echo "${result% }"
}
```

```
test_values=(0 59 3661 86400 90061)
for value in "${test_values[@]}"; do
    echo "$value: $(seconds_to_human "$value")"
done
=== Generated script ===
#!/bin/bash
seconds_to_human() {
    local total_seconds="$1"
    local result=""
    if [ "$total_seconds" -eq 0 ]; then
        echo "0s"
        return
    fi
    local hours=$((total_seconds / 3600))
    local minutes=$(((total_seconds % 3600) / 60))
    local seconds=$((total_seconds % 60))
    if [ "$hours" -gt 0 ]; then
        result="${result}${hours}h "
    fi
    if [ "$minutes" -gt 0 ]; then
        result="${result}${minutes}m "
    fi
    if [ "$seconds" -gt 0 ]; then
        result="${result}${seconds}s"
    fi
    echo "${result% }"
}
```

```
test_values=(0 59 3661 86400 90061)
for value in "${test_values[@]}"; do
  echo "$value: $(seconds_to_human "$value")"
doneREJECTED
REJECTED=== Verdict ===
REJECTED
[loop] until check: last_line='REJECTED' expected='ACCEPTED'
[loop] Iteration 1/3 \u2014 condition not met, looping back
#!/bin/bash
seconds_to_human() {
  local total_seconds="$1"
  local result=""
  if [ "$total_seconds" -eq 0 ]; then
    echo "0s"
    return
  fi
  local hours=$((total_seconds / 3600))
  local minutes=$(((total_seconds % 3600) / 60))
  local seconds=$((total_seconds % 60))
  if [ "$hours" -gt 0 ]; then
    result="${result}${hours}h "
  fi
  if [ "$minutes" -gt 0 ]; then
    result="${result}${minutes}m "
  fi
  if [ "$seconds" -gt 0 ]; then
    result="${result}${seconds}s"
```

```
    fi
    echo "${result% }"
}
test_values=(0 59 3661 86400 90061)
for value in "${test_values[@]}"; do
    echo "$value: $(seconds_to_human "$value")"
done
=== Generated script ===
#!/bin/bash
seconds_to_human() {
    local total_seconds="$1"
    local result=""
    if [ "$total_seconds" -eq 0 ]; then
        echo "0s"
        return
    fi
    local hours=$((total_seconds / 3600))
    local minutes=$(((total_seconds % 3600) / 60))
    local seconds=$((total_seconds % 60))
    if [ "$hours" -gt 0 ]; then
        result="${result}${hours}h "
    fi
    if [ "$minutes" -gt 0 ]; then
        result="${result}${minutes}m "
    fi
    if [ "$seconds" -gt 0 ]; then
        result="${result}${seconds}s"
```

```
  fi
  echo "${result% }"
}
test_values=(0 59 3661 86400 90061)
for value in "${test_values[@]}"; do
  echo "$value: $(seconds_to_human "$value")"
doneREJECTED
REJECTED=== Verdict ===
REJECTED
[loop] until check: last_line='REJECTED' expected='ACCEPTED'
[loop] Iteration 2/3 \u2014 condition not met, looping back
#!/bin/bash
seconds_to_human() {
  local total_seconds="$1"
  local result=""
  if [ "$total_seconds" -eq 0 ]; then
    echo "0s"
    return
  fi
  local hours=$((total_seconds / 3600))
  local minutes=$(((total_seconds % 3600) / 60))
  local seconds=$((total_seconds % 60))
  if [ "$hours" -gt 0 ]; then
    result="${result}${hours}h "
  fi
  if [ "$minutes" -gt 0 ]; then
    result="${result}${minutes}m "
```

```
    fi
    if [ "$seconds" -gt 0 ]; then
        result="${result}${seconds}s"
    fi
    echo "${result% }"
}
test_values=(0 59 3661 86400 90061)
for value in "${test_values[@]}"; do
    echo "$value: $(seconds_to_human "$value")"
done
=== Generated script ===
#!/bin/bash
seconds_to_human() {
    local total_seconds="$1"
    local result=""
    if [ "$total_seconds" -eq 0 ]; then
        echo "0s"
        return
    fi
    local hours=$((total_seconds / 3600))
    local minutes=$(((total_seconds % 3600) / 60))
    local seconds=$((total_seconds % 60))
    if [ "$hours" -gt 0 ]; then
        result="${result}${hours}h "
    fi
    if [ "$minutes" -gt 0 ]; then
        result="${result}${minutes}m "
```

```
    fi
    if [ "$seconds" -gt 0 ]; then
        result="${result}${seconds}s"
    fi
    echo "${result% }"
}
test_values=(0 59 3661 86400 90061)
for value in "${test_values[@]}"; do
    echo "$value: $(seconds_to_human "$value")"
doneACCEPTED
ACCEPTED=== Verdict ===
ACCEPTED
[loop] Exiting loop after 3 iteration(s). reason: until condition met
=== Final accepted script \u2014 executing ===
0: 0s
59: 59s
3661: 1h 1m 1s
86400: 24h
90061: 25h 1m 1s
/home/igor >
------------------------------------------------------------
/home/igor > Sent to mshell (1731 bytes)
Received from GUI editor:
---------------------------------------
# Pattern 8  Multi-Stage Bash + Multi-Model Pipeline
> **Edition:** Bash-Only · Art2Dec SoftLab, 2026
## What it does
```

Replaces the C/Python/Rust multi-language pipeline with a fully bash equivalent: bash

generates data with `awk`, bash computes statistics, LLM @1 analyzes, LLM @2 summarizes,

bash formats the final report.

Use case

Complex data processing workflows requiring computational processing and natural language

intelligence \u2014 no external language runtimes.

Key concept

Language blocks and LLM blocks are interchangeable pipeline stages.

Flow diagram

```

bash >raw_data   (awk: 10 pseudo-random numbers)

bash <raw_data >stats    (awk: stats)

@1 <stats >analysis      (LLM: distribution insight)

@2 <analysis >summary    (LLM: 5-word tweet)

bash <raw_data <stats <analysis <summary   (report)

```

Code

```bash >raw_data

awk 'BEGIN { srand(42); for(i=1;i<=10;i++) printf "%d%s", int(rand()*100), (i<10?",":"\n") }'

```

```bash <raw_data >stats

echo "$(cat $MSH_VAR_raw_data)" | tr ',' '\n' | awk '

  BEGIN { min=9999; max=-9999; sum=0; n=0 }

  { v=$1+0; if(v<min)min=v; if(v>max)max=v; sum+=v; n++ }
```

END { printf "count=%d sum=%d min=%d max=%d mean=%.2f\n",
n,sum,min,max,sum/n }

'

```

<!--@1 <stats >analysis

Given these statistics about a 10-number dataset, write one sentence describing

what you can infer about the spread and typical value.

-->

<!--@2 <analysis >summary

Compress this one-sentence analysis into a 5-word tweet-style summary. No hashtags.

-->

```bash <raw_data <stats <analysis <summary

echo "[raw_data] : $(cat $MSH_VAR_raw_data)"

echo "[stats]    : $(cat $MSH_VAR_stats)"

echo "[analysis] : $(cat $MSH_VAR_analysis)"

echo "[summary]  : $(cat $MSH_VAR_summary)"

```

77,52,37,62,29,83,41,89,39,39

count=10 sum=548 min=29 max=89 mean=54.80

The dataset shows a moderate spread with values ranging from 29 to 89 (a range of 60), and since the mean of 54.80 falls roughly in the middle of this range, the data appears to be reasonably well-distributed around the typical value without extreme skewness.

Moderate spread, centered around mean

[raw_data] : 77,52,37,62,29,83,41,89,39,39

[stats]   : count=10 sum=548 min=29 max=89 mean=54.80

[analysis] : The dataset shows a moderate spread with values ranging from 29 to 89 (a range of 60), and since the mean of 54.80 falls roughly in the middle of this range, the data
```

appears to be reasonably well-distributed around the typical value without extreme skewness.

[summary] : Moderate spread, centered around mean

/home/igor >

/home/igor > Sent to mshell (1433 bytes)

Received from GUI editor:

Pattern 9 Routing: LLM Classifies - Conditional Branch Executes

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

LLM @1 classifies input into `SYSINFO`, `TEXTPROC`, or `MATHCALC`. Only the matching bash branch executes. Three test cases demonstrated.

Use case

Dynamic task routing \u2014 appropriate handler depends on the nature of the input.

Key concept

`if=varname:expected_value` on any block makes it conditional. The LLM classifier must return a single predictable word.

Flow diagram

```

bash >inputN

@1 <inputN >routeN (classify)

bash <routeN if=routeN:SYSINFO (system info)

bash <routeN if=routeN:TEXTPROC (text analysis)

bash <routeN if=routeN:MATHCALC (arithmetic)

bash <routeN (print classification)

```

Code

```bash >input1
echo "What is the current system load and disk usage?"
```

<!--@1 <input1 >route1
Classify this query into exactly ONE word: SYSINFO / TEXTPROC / MATHCALC.
Reply with only that one word.
-->

```bash <route1 if=route1:SYSINFO
echo "=== SYSINFO ===" && uname -s && uptime
```

```bash <route1 if=route1:TEXTPROC
echo "=== TEXTPROC ===" && echo "Text processing handler"
```

```bash <route1 if=route1:MATHCALC
echo "=== MATHCALC ===" && echo "Math calculation handler"
```

```bash <route1
echo "\u2192 Classified as: $(cat $MSH_VAR_route1)"
```

> **Note:** See pattern-09.md for full TEXTPROC and MATHCALC test cases.

What is the current system load and disk usage?

SYSINFO

=== SYSINFO ===

Linux

09:00:29 up 17 days, 23:06, 1 user, load average: 0.25, 0.27, 0.17

\u2192 Classified as: SYSINFO

/home/igor >

/home/igor > Sent to mshell (1990 bytes)

Received from GUI editor:

Pattern 10 Full Pipeline: All Patterns Combined

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Bash generates Fibonacci numbers, bash computes statistics, LLM @1 analyzes, LLM @2

writes a poem, bash collects both, LLM @1 synthesizes, bash formats with a decorative

ASCII frame.

Use case

A reference example showing how all building blocks compose naturally.

Key concept

There is no special "combine patterns" syntax \u2014 patterns compose because every block reads

from named variables. The document structure defines the execution graph.

Flow diagram

```

bash >raw_data   (Fibonacci via bash loop)

bash <raw_data >stats    (awk: statistics)

@1 <stats >analysis      (LLM: insight)

@2 <raw_data >poem       (LLM: 2-line poem)

bash <analysis <poem     (collect)

@1 <analysis <poem >combined   (synthesize)

bash <combined          (ASCII-framed output)
```

```

## Code

```bash >raw_data
a=0; b=1; result=""
for i in $(seq 1 10); do
  result="${result}${b} "
  tmp=$((a+b)); a=$b; b=$tmp
done
echo "${result% }" | tr ' ' ','
```

```bash <raw_data >stats
echo "$(cat $MSH_VAR_raw_data)" | tr ',' '\n' | awk '
  BEGIN { sum=0; max=0; n=0; prev=0 }
  { v=$1+0; sum+=v; n++; if(v>max)max=v; if(NR>1&&prev>0)ratio=v/prev; prev=v }
  END { printf "count=%d sum=%d max=%d mean=%.1f
last_ratio=%.4f\n",n,sum,max,sum/n,ratio }
'
```

<!--@1 <stats >analysis
These are statistics about the first 10 Fibonacci numbers.
In one sentence, describe what is mathematically interesting.
-->

<!--@2 <raw_data >poem
Write a 2-line poem about Fibonacci numbers and natural growth.
-->

```bash <analysis <poem
echo "Analysis: $(cat $MSH_VAR_analysis)" && echo "" && echo "Poem:" && cat
"$MSH_VAR_poem"
```
```

```

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

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

-->

```bash <combined

text=$(cat "$MSH_VAR_combined")

border=$(printf '%0.s=' $(seq 1 60))

echo "$border" && echo "$text" | fold -w 60 && echo "$border"

```

1,1,2,3,5,8,13,21,34,55

count=10 sum=143 max=55 mean=14.3 last_ratio=1.6176

The last_ratio of 1.6176 is mathematically interesting because it represents the golden ratio (\u03c6 \u2248 1.618) that consecutive Fibonacci numbers approach as the sequence progresses, demonstrating how even within just the first 10 terms, this fundamental mathematical constant emerges naturally from the recursive structure.

From one and one to forests\u2019 climb, Fibonacci threads through leaves and time.

In spiraled shells and branching trees, numbers bloom with nature\u2019s ease.

Analysis: The last_ratio of 1.6176 is mathematically interesting because it represents the golden ratio (\u03c6 \u2248 1.618) that consecutive Fibonacci numbers approach as the sequence progresses, demonstrating how even within just the first 10 terms, this fundamental mathematical constant emerges naturally from the recursive structure.

Poem:

From one and one to forests\u2019 climb, Fibonacci threads through leaves and time.

In spiraled shells and branching trees, numbers bloom with nature\u2019s ease.The Fibonacci sequence beautifully demonstrates how mathematics and nature intertwine, as the golden ratio (\u03c6 \u2248 1.618) emerges naturally from the simple recursive pattern and manifests in spiraled shells, branching trees, and the very structure of growing forests.

==========

The Fibonacci sequence beautifully demonstrates how mathemat
```

ics and nature intertwine, as the golden ratio (\u03c6 \u2248 1.618

) emerges naturally from the simple recursive pattern and ma

nifests in spiraled shells, branching trees, and the very st

ructure of growing forests.

==========

/home/igor >

/home/igor > Sent to mshell (1315 bytes)

Received from GUI editor:

Pattern 11 MShell Node with Multiple Models

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Uses native `mshell` blocks with `ollama1`/`ollama2` alongside bash. Bash sets input

variables, mshell calls generate explanation and keywords, bash parses and formats

the output.

Use case

Workflows where mshell's native AI commands are preferable to LLM directives.

Key concept

`mshell` blocks support the same `<in_vars` and `>out_var` system as all other blocks.

Input variables are available as `$varname` inside mshell.

Flow diagram

```

bash >topic >audience

mshell <topic <audience >explanation   (ollama1)

mshell <explanation >keywords         (ollama2)

bash <explanation <keywords          (parse + digest)
```

```

## Code

```bash >topic

echo "distributed systems"

```

```bash >audience

echo "senior engineer"

```

```mshell <topic <audience >explanation

ollama1 "Explain $topic in one sentence tailored for a $audience"

```

```mshell <explanation >keywords

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

```

```bash <explanation <keywords

echo "Topic    : $(cat $MSH_VAR_topic)"

echo "Audience : $(cat $MSH_VAR_audience)"

echo "Explanation: $(cat $MSH_VAR_explanation)"

echo "Keywords : $(cat $MSH_VAR_keywords)"

```

---

distributed systems

senior engineer

Distributed systems are fundamentally about managing the trade-offs between consistency, availability, and partition tolerance (CAP theorem) while handling the complexities of network failures, data replication strategies, consensus algorithms, and the inevitability that at scale, everything will fail.consistency,availability,partition toleranceTopic   : distributed systems
```

Audience : senior engineer

Explanation: Distributed systems are fundamentally about managing the trade-offs between consistency, availability, and partition tolerance (CAP theorem) while handling the complexities of network failures, data replication strategies, consensus algorithms, and the inevitability that at scale, everything will fail.

Keywords : consistency,availability,partition tolerance

/home/igor >

/home/igor > Sent to mshell (1663 bytes)

Received from GUI editor:

Pattern 12 Sequential 3-Model Query + Synthesis

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Three LLM models answer the same question sequentially from three different perspectives

(beginner, analogy, technical). LLM @1 synthesizes all three into one sentence. Bash

formats the comparison panel.

Use case

High-quality ensemble answer generation \u2014 each model contributes a distinct perspective,

then a synthesis pass unifies them.

Key concept

Three sequential LLM calls write to independent output variables. All three are available

to the synthesis call and the final bash block.

Flow diagram

```

bash >question
```

@1 <question >ans1   (beginner perspective)

@2 <question >ans2   (real-world analogy)

@3 <question >ans3   (technical UNIX terms)

@1 <ans1 <ans2 <ans3 >final   (synthesize)

bash <ans1 <ans2 <ans3 <final (comparison panel)

```

## Code

```bash >question

echo "What is a pipe in bash and why is it one of the most powerful features of UNIX?"

```

<!--@1 <question >ans1

Explain in one sentence for someone who has never used a terminal.

-->

<!--@2 <question >ans2

Explain in one sentence using a real-world physical analogy (no computer terms).

-->

<!--@3 <question >ans3

Explain in one sentence using precise technical UNIX/bash terminology.

-->

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

Synthesize these three explanations into one perfect sentence accessible yet technically accurate.

-->

```bash <ans1 <ans2 <ans3 <final

echo "Beginner : $(cat $MSH_VAR_ans1)"

echo ""

echo "Analogy  : $(cat $MSH_VAR_ans2)"

```
echo ""

echo "Technical: $(cat $MSH_VAR_ans3)"

echo ""

echo "SYNTHESIS: $(cat $MSH_VAR_final)"
```

What is a pipe in bash and why is it one of the most powerful features of UNIX?

A pipe (|) in bash is a simple symbol that lets you chain commands together by sending the output of one program directly as input to another, making it incredibly powerful because you can combine small, specialized tools to solve complex problems without writing custom software.

A pipe in bash is like connecting the spout of one machine directly into the intake of another so whatever one produces flows instantly into the next, making it incredibly powerful for chaining many small tools into a seamless assembly line.

A pipe in bash is an inter-process communication mechanism that redirects the standard output (stdout) of one process to the standard input (stdin) of another, enabling the composition of multiple single-purpose utilities into a pipeline for efficient, file-less data streaming and transformation, which embodies the UNIX philosophy of composable tools and is foundational to its power.

A pipe (|) in bash is like connecting an assembly line where the output of one specialized tool flows directly as input to the next, enabling you to chain together simple programs into powerful workflows that process data seamlessly without temporary files\u2014embodying UNIX's philosophy of combining small, focused utilities to solve complex problems.

Beginner : A pipe (|) in bash is a simple symbol that lets you chain commands together by sending the output of one program directly as input to another, making it incredibly powerful because you can combine small, specialized tools to solve complex problems without writing custom software.

Analogy : A pipe in bash is like connecting the spout of one machine directly into the intake of another so whatever one produces flows instantly into the next, making it incredibly powerful for chaining many small tools into a seamless assembly line.

Technical: A pipe in bash is an inter-process communication mechanism that redirects the standard output (stdout) of one process to the standard input (stdin) of another, enabling the composition of multiple single-purpose utilities into a pipeline for efficient, file-less data streaming and transformation, which embodies the UNIX philosophy of composable tools and is foundational to its power.

SYNTHESIS: A pipe (|) in bash is like connecting an assembly line where the output of one specialized tool flows directly as input to the next, enabling you to chain together simple programs into powerful workflows that process data seamlessly without temporary files\u2014embodying UNIX's philosophy of combining small, focused utilities to solve complex problems.

/home/igor >

/home/igor > Sent to mshell (1630 bytes)

Received from GUI editor:

Pattern 13 WHILE Loop: Iterative Counter with LLM Commentary

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Initializes `status=running` and `counter=0`. WHILE loop runs while `status == running`.

Each iteration: bash increments counter, writes via `$MSH_VAR_*`. LLM gives one UNIX/bash

fact about the current number. At counter \u2265 3 status becomes `done`.

Use case

Any iterative processing where the exit condition depends on accumulated state.

Key concepts

- Multi-`>outvar` bash block must write to `$MSH_VAR_*` directly \u2014 stdout not captured.

- WHILE reads the **last non-empty line** of the condition variable.

- Use `running`/`done` flag \u2014 most reliable exit pattern.

Flow diagram

```

bash >status="running"

bash >counter="0"
```

[WHILE status:running]

 bash <counter <status >counter >status

 @1 <counter >fact

 bash <fact

[END_WHILE]

bash <counter

```

## Code

```bash >status
echo "running"
```

```bash >counter
echo "0"
```

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

```bash <counter <status >counter >status
val=$(cat "$MSH_VAR_counter")
val=$((val + 1))
echo "$val" > "$MSH_VAR_counter"
if [ "$val" -ge 3 ]; then
 echo "done" > "$MSH_VAR_status"
else
 echo "running" > "$MSH_VAR_status"
fi
echo "Iteration $val complete"
```
```

<!--@1 <counter >fact

The input contains a small integer. In one sentence, share an interesting fact

about this number related to UNIX, bash, or computer science.

-->

```bash <fact

echo "[iter $(cat $MSH_VAR_counter)] $(cat $MSH_VAR_fact)"

```

<!--@end_while-->

```bash <counter

echo "=== WHILE done. Final counter = $(cat $MSH_VAR_counter) ==="

```

---

running

0

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

Iteration 1 complete

The number 1 represents the exit status code in bash and UNIX systems that indicates an error or failure, making it the most common way programs signal that something went wrong, in contrast to 0 which means success.

[iter 1] The number 1 represents the exit status code in bash and UNIX systems that indicates an error or failure, making it the most common way programs signal that something went wrong, in contrast to 0 which means success.

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

Iteration 2 complete

The number 2 represents standard error (stderr) in UNIX file descriptors, which is why you use `2>` to redirect error messages in bash, alongside stdin (0) and stdout (1) forming the fundamental trio of data streams in UNIX systems.

[iter 2] The number 2 represents standard error (stderr) in UNIX file descriptors, which is why you use `2>` to redirect error messages in bash, alongside stdin (0) and stdout (1) forming the fundamental trio of data streams in UNIX systems.

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

Iteration 3 complete

The number 3 is the first file descriptor available for custom use in UNIX programs, since 0, 1, and 2 are reserved for stdin, stdout, and stderr respectively, which is why you might see `exec 3< file.txt` in bash scripts to open additional input streams.

[iter 3] The number 3 is the first file descriptor available for custom use in UNIX programs, since 0, 1, and 2 are reserved for stdin, stdout, and stderr respectively, which is why you might see `exec 3< file.txt` in bash scripts to open additional input streams.

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

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

/home/igor >

/home/igor > Sent to mshell (1128 bytes)

Received from GUI editor:

Pattern 14 FOREACH: LLM Processes Each Item in a List

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Bash creates a list of three UNIX tools with `printf`. FOREACH iterates line by line.

LLM describes each tool in one sentence. Bash prints each result in a formatted box.

Use case

Processing any list where each item needs independent LLM analysis.

Key concepts

- `printf "a\nb\nc"` for correct line-by-line FOREACH splitting.

- Iterator variable set automatically by runtime.

Flow diagram

```

bash >commands = "awk\nsed\nxargs"
```

[FOREACH cmd in commands]

@1 <cmd >description

bash <description

[END_FOREACH]

bash

```

## Code

```bash >commands

printf "awk\nsed\nxargs"

```

<!--@foreach cmd in commands-->

<!--@1 <cmd >description

The input contains the name of a UNIX command-line tool.

In one sentence, describe what it does and its single most powerful use case.

-->

```bash <description

echo "\u2501\u2501\u2501 $(cat $MSH_VAR_cmd | tr '[:lower:]' '[:upper:]')
\u2501\u2501\u2501"

cat "$MSH_VAR_description"

echo ""

```

<!--@end_foreach-->

```bash

echo "=== All commands described ==="

```

---

awk
```

sed

xargs[foreach] iter 1: cmd=awk

AWK is a pattern-scanning and data-extraction language that processes text files line by line, with its most powerful use case being the ability to effortlessly parse and manipulate structured data like CSV files or log files using built-in field separation and rich programming constructs in compact one-liners.

\u2501\u2501\u2501 AWK \u2501\u2501\u2501

AWK is a pattern-scanning and data-extraction language that processes text files line by line, with its most powerful use case being the ability to effortlessly parse and manipulate structured data like CSV files or log files using built-in field separation and rich programming constructs in compact one-liners.

[foreach] iter 2: cmd=sed

SED is a stream editor that performs text transformations on input streams without opening files in an interactive editor, with its most powerful use case being the ability to perform complex find-and-replace operations across massive files or data streams using regular expressions in a single efficient pass.

\u2501\u2501\u2501 SED \u2501\u2501\u2501

SED is a stream editor that performs text transformations on input streams without opening files in an interactive editor, with its most powerful use case being the ability to perform complex find-and-replace operations across massive files or data streams using regular expressions in a single efficient pass.

[foreach] iter 3: cmd=xargs

xargs takes input from stdin and converts it into command-line arguments for other programs, with its most powerful use case being the ability to process lists of files or data in parallel batches (using `-P`) while avoiding "argument list too long" errors when dealing with thousands of items.

\u2501\u2501\u2501 XARGS \u2501\u2501\u2501

xargs takes input from stdin and converts it into command-line arguments for other programs, with its most powerful use case being the ability to process lists of files or data in parallel batches (using `-P`) while avoiding "argument list too long" errors when dealing with thousands of items.

=== All commands described ===

/home/igor >

/home/igor > Sent to mshell (1634 bytes)

Received from GUI editor:

Pattern 15 TRY/CATCH: Safe Execution with Error Capture

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Bash initializes a metric string. TRY block uses strict `grep` regex that intentionally fails (`|| exit 1`). CATCH prints hardcoded error message. Safe fallback bash block parses with robust `awk` and prints a bar-chart metrics table.

Use case

Any pipeline where one parsing strategy may fail and a safe fallback must be guaranteed.

Key concepts

- `|| exit 1` guarantees non-zero exit on any error.
- CATCH does **not** use `<errvar` \u2014 print `"try_block_failed"` as literal.
- Pipeline continues normally after `<!--@end_try-->`.

Flow diagram

```
bash >input
[TRY]
 bash <input >result    (strict parse \u2192 || exit 1)
[CATCH >error]
 bash                (print "try_block_failed")
[END_TRY]
bash <input >safe_result (robust awk fallback)
bash <safe_result
```

Code

```bash >input
echo "system:cpu:82 system:mem:91 disk:root:45"
```

<!--@try-->

```bash <input >result
data=$(cat "$MSH_VAR_input")
echo "$data" | grep -P '^\w+=\d+$' || exit 1
```

<!--@catch >error-->

```bash
echo "=== Caught error: try_block_failed ==="
echo "Strict parser failed. Switching to robust fallback."
```

<!--@end_try-->

```bash <input >safe_result
data=$(cat "$MSH_VAR_input")
echo "=== Metrics Report ==="
echo "$data" | tr ' ' '\n' | awk -F':' '
  NF==3 {
    pct=$3+0; bar=""; filled=int(pct/5)
    for(i=0;i<filled;i++) bar=bar"\u2588"
    for(i=filled;i<20;i++) bar=bar"\u2591"
    printf "  %-15s %s %3d%%\n", $1"/"$2, bar, pct
  }
'
```

```bash <safe_result
```

cat "$MSH_VAR_safe_result"

```

---

system:cpu:82 system:mem:91 disk:root:45

[try] executing try block

[try] try block failed, executing catch block

=== Caught error: try_block_failed ===

Strict parser failed. Switching to robust fallback.

=== Metrics Report ===

  system/cpu
\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u258
8\u2588\u2588\u2588\u2588\u2591\u2591\u2591\u2591  82%

  system/mem
\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u258
8\u2588\u2588\u2588\u2588\u2588\u2588\u2591\u2591  91%

  disk/root
\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2591\u2591\u259
1\u2591\u2591\u2591\u2591\u2591\u2591\u2591\u2591  45%

=== Metrics Report ===

  system/cpu
\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u258
8\u2588\u2588\u2588\u2588\u2591\u2591\u2591\u2591  82%

  system/mem
\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u258
8\u2588\u2588\u2588\u2588\u2588\u2588\u2591\u2591  91%

  disk/root
\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2591\u2591\u259
1\u2591\u2591\u2591\u2591\u2591\u2591\u2591\u2591  45%

/home/igor >

/home/igor > Sent to mshell (1586 bytes)

Received from GUI editor:

---
```

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

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Bash creates a two-line server log dataset. SPLIT marks the logical division into `dataset_1` and `dataset_2`. Two sequential LLMs analyze each part. LLM @1 synthesizes both analyses. Bash prints the unified report.

Use case

Sequential analysis of time-partitioned data (e.g. morning vs. evening shifts) followed by synthesis.

Key concepts

- SPLIT and MERGE are **visual markers** \u2014 they execute no code.
- Sequential LLM calls guarantee variable availability for the synthesis step.

Flow diagram

```
bash >dataset
[SPLIT dataset into 2]  \u2192  dataset_1, dataset_2
  @1 <dataset_1 >analysis1
  @2 <dataset_2 >analysis2
[MERGE]
  @1 <analysis1 <analysis2 >combined
bash <analysis1 <analysis2 <combined
```

Code

```bash >dataset
printf "morning: GET /api/users 12400req avg=45ms errors=12\nevening: GET /api/users 31800req avg=112ms errors=187"
```

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

<!--@1 <dataset_1 >analysis1

The input is a server log summary. In one sentence, characterize performance and errors.

-->

<!--@2 <dataset_2 >analysis2

The input is a server log summary. In one sentence, characterize performance and errors.

-->

<!--@merge-->

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

Synthesize these two performance analyses into a two-sentence operational summary.

-->

```bash <analysis1 <analysis2 <combined

echo "Morning: $(cat $MSH_VAR_analysis1)"

echo "Evening: $(cat $MSH_VAR_analysis2)"

echo "Summary: $(cat $MSH_VAR_combined)"

```

---

morning: GET /api/users 12400req avg=45ms errors=12

evening: GET /api/users 31800req avg=112ms errors=187[split] dataset_1 = morning: GET /api/users 12400req avg=45ms errors=12

[split] dataset_2 = evening: GET /api/users 31800req avg=112ms errors=187

The morning traffic shows healthy performance with 12,400 requests averaging 45ms response time and an excellent error rate of just 0.097% (12 errors), indicating the system is handling load well with minimal issues.

Evening traffic is moderate with an acceptable average response time of 112 ms, but 187 errors indicate a nontrivial reliability issue that needs investigation.

The system demonstrates strong morning performance with 12,400 requests averaging 45ms and only a 0.097% error rate, but evening traffic shows degraded reliability with 187 errors and slower 112ms response times. This pattern suggests the system may be

experiencing increased load or resource constraints during peak evening hours that require investigation and potentially additional capacity.

Morning: The morning traffic shows healthy performance with 12,400 requests averaging 45ms response time and an excellent error rate of just 0.097% (12 errors), indicating the system is handling load well with minimal issues.

Evening: Evening traffic is moderate with an acceptable average response time of 112 ms, but 187 errors indicate a nontrivial reliability issue that needs investigation.

Summary: The system demonstrates strong morning performance with 12,400 requests averaging 45ms and only a 0.097% error rate, but evening traffic shows degraded reliability with 187 errors and slower 112ms response times. This pattern suggests the system may be experiencing increased load or resource constraints during peak evening hours that require investigation and potentially additional capacity.

/home/igor >

/home/igor > Sent to mshell (1501 bytes)

Received from GUI editor:

Pattern 17 CONFIG Node: Parameterized Pipeline

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

CONFIG documents `subject`, `style`, `max_words`. Bash blocks set runtime values.

LLM @1 explains the topic in style. LLM @2 extracts 5 keywords. Bash formats the

report with `printf`.

Use case

Reusable pipeline templates \u2014 change two bash `echo` lines to switch topic and style.

Key concepts

- CONFIG is **documentation only** \u2014 always pair with bash `echo` blocks.

- Avoid duplicating output with an extra `cat` \u2014 mshell captures stdout.

Flow diagram

```
```

config (subject, style, max_words \u2014 docs only)

bash >subject >style

@1 <subject <style >explanation

@2 <explanation >keywords

bash <explanation <keywords   (formatted report)

```

## Code

```config

subject=black holes

style=poetic and accessible

max_words=50

```

```bash >subject

echo "black holes"

```

```bash >style

echo "poetic and accessible"

```

<!--@1 <subject <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.

-->

```bash <explanation <keywords

echo "Subject  : $(cat $MSH_VAR_subject)"

echo "Style    : $(cat $MSH_VAR_style)"

echo "Words    : $(cat $MSH_VAR_explanation | wc -w)"

echo ""

echo "Explanation:" && cat "$MSH_VAR_explanation"

echo "" && echo "Keywords: $(cat $MSH_VAR_keywords)"

```

[config] subject=black holes

[config] style=poetic and accessible

[config] max_words=50

black holes

poetic and accessible

Where light itself cannot escape, darkness devours even starlight's whisper. These cosmic wells bend space and time into spirals, pulling galaxies into their silent dance. At their edge, moments stretch into eternity\u2014nature's most mysterious guardians of secrets we may never unlock.

black holes, darkness, spacetime, galaxies, secrets

Subject : black holes

Style   : poetic and accessible

Words   : 41

Explanation:

Where light itself cannot escape, darkness devours even starlight's whisper. These cosmic wells bend space and time into spirals, pulling galaxies into their silent dance. At their edge, moments stretch into eternity\u2014nature's most mysterious guardians of secrets we may never unlock.

Keywords: black holes, darkness, spacetime, galaxies, secrets

/home/igor >

/home/igor > Sent to mshell (1489 bytes)
```

Received from GUI editor:

Pattern 18 FOREACH + Multi-Model: Sequential Batch Processing

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Bash creates a list of three bash concepts. FOREACH iterates. Per item: @1 explains for beginner, @2 gives analogy. Both run sequentially. Bash formats each result in a box.

Use case

Batch processing where each item needs multiple LLM perspectives.

Key concepts

- Sequential LLM calls inside FOREACH are reliable and guaranteed to work.
- Each LLM call's output variable is available immediately to subsequent blocks.
- `tr '_' ' '` converts underscore-names to readable titles.

Flow diagram

```
bash >topics = "process_substitution\nbrace_expansion\nhere_documents"
[FOREACH topic in topics]
  @1 <topic >explanation   (beginner explanation)
  @2 <topic >analogy       (real-world analogy)
  bash <explanation <analogy
[END_FOREACH]
bash
```

Code

```bash >topics
printf "process_substitution\nbrace_expansion\nhere_documents"
```

<!--@foreach topic in topics-->

<!--@1 <topic >explanation

The input is a bash concept name. Explain it in one sentence for a beginner.

-->

<!--@2 <topic >analogy

The input is a bash concept name. Give one real-world analogy. One sentence only.

-->

```bash <explanation <analogy

echo "\u250c\u2500\u2500\u2500 $(cat $MSH_VAR_topic | tr '_' ' ')
\u2500\u2500\u2500\u2510"

echo "  $(cat $MSH_VAR_explanation)"

echo "  \u21b3 $(cat $MSH_VAR_analogy)"

echo ""

```

<!--@end_foreach-->

```bash

echo "=== Batch complete ==="

```

---

process_substitution

brace_expansion

here_documents[foreach] iter 1: topic=process_substitution

Process substitution uses `<(command)` or `>(command)` to treat the output or input of a command as if it were a temporary file, allowing you to pass command output directly to programs that normally only accept filenames rather than piped input.

Process substitution is like temporarily routing the output of a live radio broadcast into a device that only accepts recorded tapes, by pretending the live stream is a tape just long enough to use it.

\u250c\u2500\u2500\u2500 process substitution \u2500\u2500\u2500\u2510

Process substitution uses `<(command)` or `>(command)` to treat the output or input of a command as if it were a temporary file, allowing you to pass command output directly to programs that normally only accept filenames rather than piped input.

\u21b3 Process substitution is like temporarily routing the output of a live radio broadcast into a device that only accepts recorded tapes, by pretending the live stream is a tape just long enough to use it.

[foreach] iter 2: topic=brace_expansion

Brace expansion uses curly braces with comma-separated values like `{a,b,c}` or ranges like `{1..5}` to automatically generate multiple arguments or filenames, so `echo file{1,2,3}.txt` becomes `echo file1.txt file2.txt file3.txt`.

Brace expansion is like writing a mailing label with optional parts in curly brackets\u2014such as \u201cApartment {1,2,3}B\u201d\u2014and having it automatically expand into separate full addresses for each choice.

\u250c\u2500\u2500\u2500 brace expansion \u2500\u2500\u2500\u2510

Brace expansion uses curly braces with comma-separated values like `{a,b,c}` or ranges like `{1..5}` to automatically generate multiple arguments or filenames, so `echo file{1,2,3}.txt` becomes `echo file1.txt file2.txt file3.txt`.

\u21b3 Brace expansion is like writing a mailing label with optional parts in curly brackets\u2014such as \u201cApartment {1,2,3}B\u201d\u2014and having it automatically expand into separate full addresses for each choice.

[foreach] iter 3: topic=here_documents

Here documents use `<<` followed by a delimiter (like `<<EOF`) to pass multiple lines of text directly to a command without needing separate files, ending when the same delimiter appears alone on a line.

A here-document is like handing a chef a written recipe enclosed between two clearly marked lines, so they know exactly where the instructions start and end and can follow them without asking further questions.

\u250c\u2500\u2500\u2500 here documents \u2500\u2500\u2500\u2510

Here documents use `<<` followed by a delimiter (like `<<EOF`) to pass multiple lines of text directly to a command without needing separate files, ending when the same delimiter appears alone on a line.

\u21b3 A here-document is like handing a chef a written recipe enclosed between two clearly marked lines, so they know exactly where the instructions start and end and can follow them without asking further questions.

=== Batch complete ===

/home/igor >

/home/igor > Sent to mshell (2003 bytes)

Received from GUI editor:

Pattern 19 WHILE Quality Gate: Generate Until Threshold

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

WHILE loop generates bash one-liners via LLM @1. LLM @2 scores each on

correctness/conciseness 1\u201310. Loop exits when score \u2265 8. Bash strips whitespace with

`tr` before numeric comparison.

Use case

Quality-driven generation \u2014 keep trying until a threshold is met.

Key concepts

- Scorer must return a bare integer \u2014 use `Reply with ONLY the integer`.

- `tr -d '[:space:]'` cleans LLM response before `[ "$sc" -ge 8 ]`.

- All variables must be initialized before the loop.

Flow diagram

```

bash >task >status="running" >iteration="0" >score="0" >snippet=""

[WHILE status:running]

  bash <iteration >iteration

  @1 <task >snippet          (generate one-liner)

  @2 <snippet >score          (rate 1-10)
```

bash <iteration <score <snippet >status

[END_WHILE]

bash <snippet <score

```

## Code

```bash >task

echo "Write a single bash one-liner that lists the 5 largest files in the current directory tree, sorted by size descending."

```

```bash >status

echo "running"

```

```bash >iteration

echo "0"

```

```bash >score

echo "0"

```

```bash >snippet

echo ""

```

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

```bash <iteration >iteration

val=$(cat "$MSH_VAR_iteration"); echo "$((val + 1))"

```

<!--@1 <task >snippet

Write ONLY the one-liner bash command. No explanation, no markdown, no shell prompt symbol.
```

-->

<!--@2 <snippet >score

Rate this bash one-liner 1-10 for correctness, conciseness, and bash idiom quality.

Reply with ONLY the integer score, nothing else.

-->

```bash <iteration <score <snippet >status

sc=$(cat "$MSH_VAR_score" | tr -d '[:space:]')

echo "[Iter $(cat $MSH_VAR_iteration)] Score=${sc}: $(cat $MSH_VAR_snippet)"

if [ "$sc" -ge 8 ] 2>/dev/null; then echo "done"; else echo "running"; fi

```

<!--@end_while-->

```bash <snippet <score

echo "=== Accepted (score=$(cat $MSH_VAR_score | tr -d '[:space:]')) ==="

cat "$MSH_VAR_snippet"

```

Write a single bash one-liner that lists the 5 largest files in the current directory tree, sorted by size descending.

running

0

0

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

1

find . -type f -exec ls -la {} + | sort -k5 -nr | head -5

9

[Iter 1] Score=9: find . -type f -exec ls -la {} + | sort -k5 -nr | head -5

done

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

=== Accepted (score=9) ===

find . -type f -exec ls -la {} + | sort -k5 -nr | head -5

/home/igor >

/home/igor > Sent to mshell (2321 bytes)

Received from GUI editor:

Pattern 20 SPLIT + MERGE: Sequential Map-Reduce Pipeline

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Bash stores a 5-sentence software engineering text. Three bash blocks split into sentence chunks with `awk`. Three sequential LLM @1 calls extract the core principle (3 words each). LLM @2 synthesizes all three into one theme sentence.

Use case

Processing long text in parallel chunks then reducing to a summary.

Key concepts

- Map phase: three sequential LLM calls, each reading a different sentence chunk.
- Reduce phase: one LLM receives all three analyses via multiple `<invar` with `[varname]:` labels.
- SPLIT and MERGE are visual markers only \u2014 they execute no code.

Flow diagram

```

bash >raw_text

bash <raw_text >sent1 / >sent2 / >sent3   (awk split)

  @1 <sent1 >analysis1   MAP: extract principle

  @1 <sent2 >analysis2
```

@1 <sent3 >analysis3

[MERGE]

@2 <analysis1 <analysis2 <analysis3 >summary   (REDUCE)

bash <analysis1 <analysis2 <analysis3 <summary

```

## Code

```bash >raw_text

echo "Clean code is not written for machines but for human beings who must maintain it. Naming matters more than any other single decision in software design. Functions should do one thing, do it well, and do it only. Tests are not overhead \u2014 they are the specification of the system. Refactoring without tests is flying blind in a thunderstorm."

```

```bash <raw_text >sent1

cat "$MSH_VAR_raw_text" | awk -F'. ' '{print $1}'

```

```bash <raw_text >sent2

cat "$MSH_VAR_raw_text" | awk -F'. ' '{print $2". "$3}'

```

```bash <raw_text >sent3

cat "$MSH_VAR_raw_text" | awk -F'. ' '{print $4". "$5}'

```

<!--@1 <sent1 >analysis1

State the core software engineering principle in 3 words max.

-->

<!--@1 <sent2 >analysis2

State the core software engineering principle in 3 words max.

-->

<!--@1 <sent3 >analysis3
```

State the core software engineering principle in 3 words max.

-->

<!--@merge-->

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

Synthesize these three principle labels into one coherent theme sentence.

-->

```bash <analysis1 <analysis2 <analysis3 <summary

echo "=== MAP ===" && echo "  $(cat $MSH_VAR_analysis1)" && echo "  $(cat $MSH_VAR_analysis2)" && echo "  $(cat $MSH_VAR_analysis3)"

echo "" && echo "=== REDUCE ===" && cat "$MSH_VAR_summary"

```

Clean code is not written for machines but for human beings who must maintain it. Naming matters more than any other single decision in software design. Functions should do one thing, do it well, and do it only. Tests are not overhead \u2014 they are the specification of the system. Refactoring without tests is flying blind in a thunderstorm.

Clea

cod. i

no. writte

Clean, simple code.

Write readable code.

Don't repeat yourself.

Write clean, simple, and readable code that avoids unnecessary repetition.

=== MAP ===

Clean, simple code.

Write readable code.

Don't repeat yourself.

=== REDUCE ===

Write clean, simple, and readable code that avoids unnecessary repetition.

/home/igor >

/home/igor > Sent to mshell (1621 bytes)

Received from GUI editor:

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

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

LOOP runs up to 3 times. LLM @1 generates a bash script. TRY executes it with

`bash "$MSH_VAR_code"`. On success writes `ok`. On failure CATCH writes `fail`.

Loop exits when `result == ok`.

Use case

Robust script generation with automatic retry on failure.

Key concepts

- CATCH does **not** use `<last_error` \u2014 writes `fail` directly.

- `result` initialized to `"fail"` before loop.

- LOOP reads last non-empty line of `result` for `until=` check.

Flow diagram

```

bash >task >result="fail" >last_error="none"

[LOOP max=3 until=result:ok]

  @1 <task <last_error >code

  bash <code

  [TRY]

    bash <code >result
```

[CATCH >last_error]

bash >result

[END_TRY]

[END_LOOP]

bash <result

```

## Code

```bash >task

echo "Write a bash script that reads /etc/hosts, counts non-comment non-empty lines, and prints: 'Active entries: N'."

```

```bash >result

echo "fail"

```

```bash >last_error

echo "none"

```

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

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

Return ONLY a pure bash script. No markdown fences. No explanation.

If the second input is not "none", fix the previous error.

-->

```bash <code

echo "=== Generated ===" && cat "$MSH_VAR_code" && echo ""

```

<!--@try-->

```bash <code >result

bash "$MSH_VAR_code"

echo "ok"

```

<!--@catch >last_error-->

```bash >result

echo "=== Error: try_block_failed ==="

echo "fail"

```

<!--@end_try-->

<!--@end_loop-->

```bash <result

echo "=== Final status: $(cat $MSH_VAR_result) ==="

```

Write a bash script that reads /etc/hosts, counts non-comment non-empty lines, and prints: 'Active entries: N'.

fail

none

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

#!/bin/bash

count=$(grep -v '^#' /etc/hosts | grep -v '^[[:space:]]*$' | wc -l)

echo "Active entries: $count"

=== Generated ===

#!/bin/bash

count=$(grep -v '^#' /etc/hosts | grep -v '^[[:space:]]*$' | wc -l)

echo "Active entries: $count"

[try] executing try block
```

Active entries: 26

ok

[try] try block succeeded

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

=== Final status: Active entries: 26

ok ===

/home/igor >

/home/igor > Sent to mshell (2178 bytes)

Received from GUI editor:

Pattern 22 Multi-Variable Output: Structured Field Extraction

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Bash stores a system alert. LLM @1 responds with strict 3-line format (`SEVERITY:`, `SERVICE:`, `ACTION:`). A bash block with three `>outvar` extracts each field via `grep -oP` and writes directly to `$MSH_VAR_*`. LLM @2 generates a runbook recommendation. Bash prints the incident card.

Use case

Structured field extraction from LLM prose output without Python.

Key concepts

- Multiple `>outvar` on CODE block: must write to `$MSH_VAR_*` directly.
- `grep -oP` replaces Python regex for field extraction.
- `MSH_VAR_*` environment variables are pre-set by the parser.

Flow diagram

```

bash >input
```

@1 <input >raw_response   (structured 3-line format)

bash <raw_response >severity >service >action

  (grep fields; write via echo > "$MSH_VAR_*")

@2 <severity <service >recommendation

bash <recommendation   (incident card)

```

## Code

```bash >input

echo "ALERT: Response time on payment-gateway spiked to 4200ms. Error rate 12.4%. Circuit breaker is OPEN."

```

<!--@1 <input >raw_response

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

SEVERITY: one word \u2014 LOW / MEDIUM / HIGH / CRITICAL

SERVICE: one word \u2014 the affected service name

ACTION: one sentence \u2014 the single most important immediate action

-->

```bash <raw_response >severity >service >action

text=$(cat "$MSH_VAR_raw_response")

echo "$(echo "$text" | grep -oP 'SEVERITY:\s*\K\S+')" > "$MSH_VAR_severity"

echo "$(echo "$text" | grep -oP 'SERVICE:\s*\K\S+')"  > "$MSH_VAR_service"

echo "$(echo "$text" | grep -oP 'ACTION:\s*\K.+')"    > "$MSH_VAR_action"

echo "Extracted: SEVERITY=$(cat $MSH_VAR_severity) SERVICE=$(cat $MSH_VAR_service)"

```

<!--@2 <severity <service >recommendation

The first input is an alert severity. The second is a service name.

Write one sentence: the recommended runbook step for this severity and service.
```

-->

```bash <recommendation

echo "SEVERITY : $(cat $MSH_VAR_severity)"

echo "SERVICE  : $(cat $MSH_VAR_service)"

echo "ACTION   : $(cat $MSH_VAR_action)"

echo "RUNBOOK  : $(cat $MSH_VAR_recommendation)"

```

ALERT: Response time on payment-gateway spiked to 4200ms. Error rate 12.4%. Circuit breaker is OPEN.

SEVERITY: CRITICAL

SERVICE: payment-gateway

ACTION: Immediately investigate the root cause of the 4200ms response time spike while the circuit breaker protects downstream services, and prepare to failover to backup payment processing if available.

Extracted: SEVERITY=CRITICAL SERVICE=payment-gateway

Immediately disable affected payment flows, verify transaction integrity and error rates, and begin rollback or failover to a known-good payment-gateway version while notifying on-call and compliance stakeholders.

SEVERITY : CRITICAL

SERVICE : payment-gateway

ACTION   : Immediately investigate the root cause of the 4200ms response time spike while the circuit breaker protects downstream services, and prepare to failover to backup payment processing if available.

RUNBOOK : Immediately disable affected payment flows, verify transaction integrity and error rates, and begin rollback or failover to a known-good payment-gateway version while notifying on-call and compliance stakeholders.

/home/igor >

/home/igor > Sent to mshell (2467 bytes)

Received from GUI editor:

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

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

CONFIG documents parameters. Bash sets runtime values. WHILE loop: @1 generates a

3-sentence explanation, @2 scores 1\u201310 with a quality bar chart rendered by bash, loop

exits at quality \u2265 7. @3 polishes for publication. Bash prints the formatted final result.

Use case

Fully adaptive quality-driven pipeline \u2014 reusable template for any topic/audience

combination.

Key concepts

- @2 receives `<target_audience` explicitly.

- Three model roles: generator (@1), scorer (@2), finisher (@3).

- Quality bar chart built with bash loop and `\u2588`/`\u2591` characters.

Flow diagram

```

config + bash >subject >target_audience >status >iteration >quality >explanation

[WHILE status:running]

  bash <iteration >iteration

  @1 <subject <target_audience >explanation

  @2 <explanation <target_audience >quality

  bash <iteration <quality >status

[END_WHILE]

@3 <explanation >final_polish

bash <final_polish <quality <iteration

```

Code

```config
subject=the CAP theorem in distributed systems
target_audience=junior developer
quality_threshold=7
```

```bash >subject
echo "the CAP theorem in distributed systems"
```

```bash >target_audience
echo "junior developer"
```

```bash >status
echo "running"
```

```bash >iteration
echo "0"
```

```bash >quality
echo "0"
```

```bash >explanation
echo ""
```

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

```bash <iteration >iteration
val=$(cat "$MSH_VAR_iteration"); echo "$((val + 1))"
```

```

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

Explain the subject to the target audience in exactly 3 sentences. No jargon.

-->

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

Rate this explanation 1-10 for clarity, accuracy, engagement. Reply with ONLY the integer.

-->

```bash <iteration <quality >status

q=$(cat "$MSH_VAR_quality" | tr -d '[:space:]')

bar=""; for i in $(seq 1 10); do [ "$i" -le "$q" ] && bar="${bar}\u2588" ||
bar="${bar}\u2591"; done

echo "[Iter $(cat $MSH_VAR_iteration)] ${q}/10 [${bar}]"

if [ "$q" -ge 7 ] 2>/dev/null; then echo "done"; else echo "running"; fi

```

<!--@end_while-->

<!--@3 <explanation >final_polish

Polish this explanation for final publication. Keep 3 sentences. No markdown.

-->

```bash <final_polish <quality <iteration

echo "Score=$(cat $MSH_VAR_quality | tr -d '[:space:]')/10  Iters=$(cat
$MSH_VAR_iteration)"

echo "" && cat "$MSH_VAR_final_polish"

```

[config] subject=the CAP theorem in distributed systems

[config] target_audience=junior developer

[config] quality_threshold=7

the CAP theorem in distributed systems
```

junior developer

running

0

0

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

1

The CAP theorem states that any distributed system can only guarantee two out of three things: Consistency (all nodes see the same data at the same time), Availability (the system keeps working even if some parts fail), and Partition tolerance (the system continues operating even when network connections between servers are broken). For example, when your database is split across multiple servers and the network connection fails, you must choose between keeping the data perfectly synchronized (consistency) or allowing the system to keep accepting requests (availability). This forces you to make important design decisions about what your application prioritizes when things go wrong.

9

[Iter 1] 9/10 [\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2591]

done

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

The CAP theorem dictates that a distributed system can simultaneously guarantee only two of three properties: consistency (all nodes observe identical data at any moment), availability (every request receives a response), and partition tolerance (operation during network failures). When a partition occurs, such as severed communication between database nodes, the system must choose between consistency\u2014halting operations to prevent stale data\u2014or availability\u2014continuing service with potential data divergence. This fundamental constraint compels architects to intentionally prioritize specific guarantees based on application requirements during infrastructure failures.

Score=9/10 Iters=1

The CAP theorem dictates that a distributed system can simultaneously guarantee only two of three properties: consistency (all nodes observe identical data at any moment), availability (every request receives a response), and partition tolerance (operation during network failures). When a partition occurs, such as severed communication between database nodes, the system must choose between consistency\u2014halting operations to prevent stale data\u2014or availability\u2014continuing service with potential data divergence. This fundamental constraint compels architects to intentionally prioritize specific guarantees based on application requirements during infrastructure failures./home/igor >

/home/igor >

/home/igor > Sent to mshell (2030 bytes)

Received from GUI editor:

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

> **Edition:** Bash-Only · Art2Dec SoftLab, 2026

What it does

Bash creates a list of log lines (two valid, one broken). FOREACH iterates. Per item:

TRY parses with `awk` + strict validation (`|| exit 1`). On success, LLM describes the

event. On failure, CATCH prints `[ERR]` as literal. Loop continues regardless.

Use case

Production log batch processing where one malformed record must never halt the pipeline.

Key concepts

- TRY/CATCH inside FOREACH = **per-item error isolation**.

- CATCH does **not** use `<parse_error` \u2014 prints literal string.

- Do not depend on variables written inside TRY \u2014 they may not exist if TRY failed.

Flow diagram

```

bash >items = "valid_log1\nbroken_log\nvalid_log2"

[FOREACH item in items]

  [TRY]

    bash <item >parsed    (awk strict parse \u2192 may exit 1)

    @1 <parsed >insight   (LLM: describe event)

    bash <insight         (print [OK])

  [CATCH >parse_error]

    bash                  (print [ERR] literal)
```

[END_TRY]

[END_FOREACH]

bash

```

## Code

```bash >items

printf '2024-01-15T14:32:01Z INFO  api-gateway status=200 duration=45ms\nINFO broken-no-timestamp status=200\n2024-01-15T14:32:05Z ERROR payment-svc status=500 duration=1240ms'

```

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

<!--@try-->

```bash <item >parsed

line=$(cat "$MSH_VAR_item")

ts=$(echo "$line" | awk '{print $1}')

echo "$ts" | grep -qE '^[0-9]{4}-[0-9]{2}-[0-9]{2}T' || exit 1

svc=$(echo "$line" | awk '{print $3}')

[ -z "$svc" ] && exit 1

echo "timestamp=$ts service=$svc $(echo $line | grep -oE 'status=[0-9]+')"

```

<!--@1 <parsed >insight

The input is a structured log entry. In one sentence, describe the event and whether it indicates a problem.

-->

```bash <insight

echo "[OK] $(cat $MSH_VAR_insight)"

```

<!--@catch >parse_error-->
```

```bash
echo "[ERR] Parse failed: try_block_failed"
```

<!--@end_try-->

<!--@end_foreach-->

```bash
echo "=== Batch complete. Errors isolated, pipeline never stopped. ==="
```

2024-01-15T14:32:01Z INFO api-gateway status=200 duration=45ms

INFO broken-no-timestamp status=200

2024-01-15T14:32:05Z ERROR payment-svc status=500 duration=1240ms[foreach] iter 1: item=2024-01-15T14:32:01Z INFO api-gateway status=200 duration=45ms

[try] executing try block

timestamp=2024-01-15T14:32:01Z service=api-gateway status=200

This log entry shows a successful API gateway request at 2:24 PM UTC on January 15th, 2024 with a 200 OK status code, indicating normal, healthy operation with no problems detected.

[OK] This log entry shows a successful API gateway request at 2:24 PM UTC on January 15th, 2024 with a 200 OK status code, indicating normal, healthy operation with no problems detected.

[try] try block succeeded

[foreach] iter 2: item=INFO broken-no-timestamp status=200

[try] executing try block

This log entry shows a successful API gateway request at 2:32 PM UTC on January 15th, 2024 with a 200 OK status code, indicating normal, healthy operation with no problems detected.

[try] try block failed, executing catch block

[ERR] Parse failed: try_block_failed

[foreach] iter 3: item=2024-01-15T14:32:05Z ERROR payment-svc status=500 duration=1240ms

[try] executing try block

timestamp=2024-01-15T14:32:05Z service=payment-svc status=500

This log entry shows a payment service failure at 2:32 PM UTC on January 15th, 2024 with a 500 Internal Server Error status code, indicating a serious problem that requires immediate investigation since payment processing is critical functionality.

[OK] This log entry shows a payment service failure at 2:32 PM UTC on January 15th, 2024 with a 500 Internal Server Error status code, indicating a serious problem that requires immediate investigation since payment processing is critical functionality.

[try] try block succeeded

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

/home/igor >

References:

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