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

> Pure Python language Edition for mshell Workflow — Art2Dec SoftLab, March 17th 2026
>
> **Pure Edition** Python-Only · No C, C++, Rust, Go, Lua or Bash — only `Python`, `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 is the Python-only edition. Every code block uses Python. No Bash, C, Rust, Go, or Lua. All data generation, processing, file I/O, iteration logic, and orchestration is done in Python. This makes mshell accessible to anyone who knows Python — no polyglot knowledge required.

Core Concepts

Variable System

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

```
`python >varname`     — produce output into a named variable
`python <varname`     — consume variable from a previous block
`python <v1 <v2`    — multiple input variables
```

Variables persist as files in `/tmp/mshell_ctx_PID/` and are accessible across all Python blocks via `os.environ['MSH_VAR_varname']`.

Reading a variable in Python:

```python
import os
value = open(os.environ['MSH_VAR_myvar']).read().strip()
```

Writing to multiple output variables directly:

```python
import os
with open(os.environ['MSH_VAR_result'], 'w') as f:
    f.write("computed value")
```

LLM Directives

```
<!--@1 <invar >outvar
Your prompt here. Use {variable} references to clarify.
-->
```

- `@1`, `@2`, `@3` address up to 3 configured models simultaneously
- With one `<invar`: variable contents are injected without a label
- With multiple `<invar`: each variable is injected with a `[varname]:` label

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, ollama, etc.), configured independently per model slot. Models can run: - **Synchronously** — blocking, sequential execution - **Asynchronously** — parallel execution with an `await` barrier for synchronization

Conditional Execution

```
`python <route if=route:SCIENCE`   — runs only if variable "route" contains "
SCIENCE"
```

Async + Await

```
<!--@1 <data >result async-->    — launch in background
`python await=result1,result2`   — block until all named variables are writte
n
```

Advanced Nodes

Node	Syntax	Semantics
WHILE	`<!--@while var:value-->` … `<!--@end_while-->`	Loop while var == value
FOREACH	`<!--@foreach item in list-->` … `<!--@end_foreach-->`	Iterate over lines of variable
TRY/CATCH	`<!--@try-->` … `<!--@catch >e-->` … `<!--@end_try-->`	Error isolation
SPLIT	`<!--@split var into N-->`	Creates var_1, var_2, … from lines
MERGE	`<!--@merge-->`	Visual reduce marker
CONFIG	` ```config … ``` `	Documentation block (no runtime injection)
LOOP	`<!--@loop max=N until=var:val-->` … `<!--@end_loop-->`	Bounded loop with exit condition

Key Rules for Advanced Nodes

- **FOREACH list:** create with `print("a\nb\nc")` — one item per line
- **WHILE condition:** body must modify the condition variable every iteration
- **CONFIG:** does not inject variables — always pair with Python `print` blocks
- **CATCH `>errvar`:** variable receives literal string `"try_block_failed"`. Never read `<errvar` inside CATCH
- **Multiple `>outvar` on CODE block:** block must write to `$MSH_VAR_*` files directly; stdout is NOT captured
- **Async jobs:** each async call must have exactly one `>outvar`

Part I — Foundational Patterns (1–12)

Pattern 1 — Linear Data Pipeline

What it does: Sequential data flow through a chain of Python transformation stages. Simulates a multi-stage ETL pipeline entirely in Python: generate → validate → transform → format → report.

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

Flow diagram:

```
python >raw_numbers
python <raw_numbers >stats
python <stats >filtered
python <filtered >formatted
python <formatted >report
python <report  (print)
```

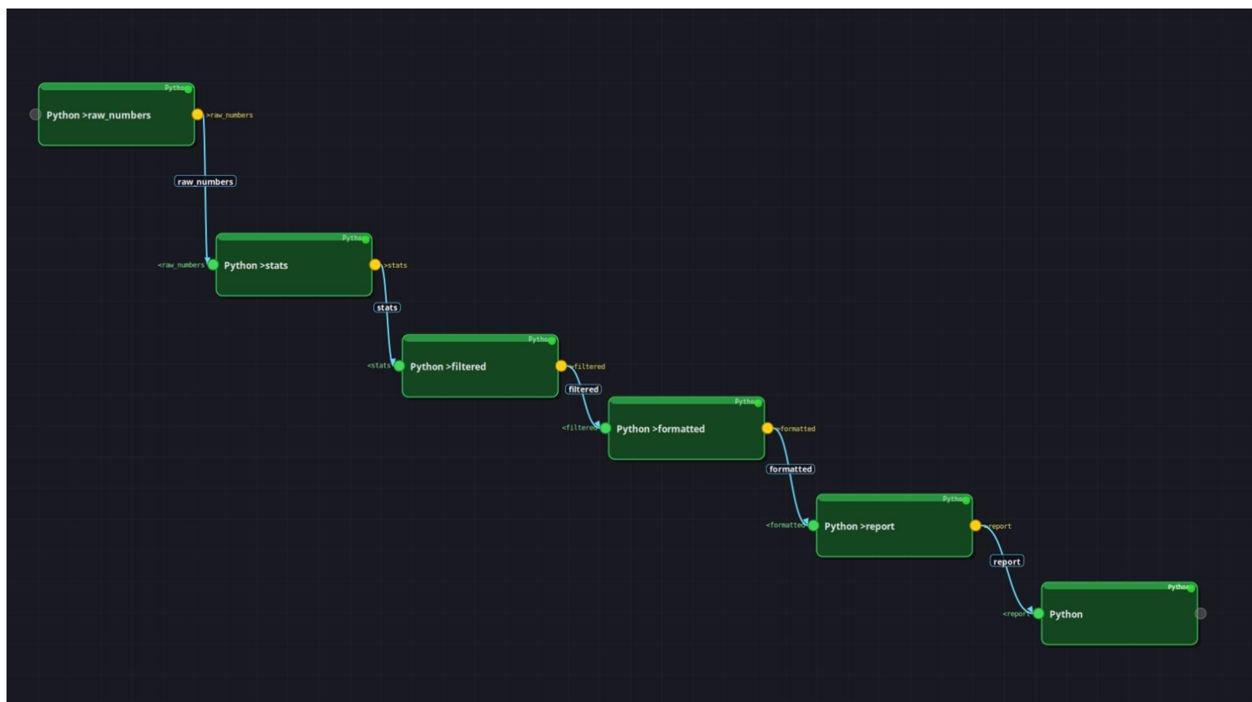
Code:

```
python >raw_numbers import random random.seed(42) numbers =
[random.randint(1, 100) for _ in range(12)] print(','.join(map(str,
numbers)))
```

```
python <raw_numbers >stats import os data =
open(os.environ['MSH_VAR_raw_numbers']).read().strip() nums = list(map(int,
data.split(','))) mean = sum(nums) / len(nums) variance = sum((x - mean) ** 2
for x in nums) / len(nums) std = variance ** 0.5 print(f"count={len(nums)}
sum={sum(nums)} mean={mean:.2f} std={std:.2f} min={min(nums)}
max={max(nums)}")
```

```
python <stats >filtered import os stats =
open(os.environ['MSH_VAR_stats']).read().strip() parts = dict(p.split('=')
for p in stats.split()) mean = float(parts['mean']) std = float(parts['std'])
```

```
print(f"Threshold analysis: values beyond mean±1σ = [{mean - std:.1f}, {mean
+ std:.1f}]") print(f"Distribution: {stats}")

python <filtered >formatted import os text =
open(os.environ['MSH_VAR_filtered']).read().strip() lines = text.split('\n')
formatted = '\n'.join(f"  → {line}" for line in lines) print("=== Pipeline
Stage 4: Formatted Report ===") print(formatted)

python <formatted >report import os content =
open(os.environ['MSH_VAR_formatted']).read().strip() report = f"""
╔══════════════════════════════════╗ ║   DATA PIPELINE FINAL REPORT    ║
╚══════════════════════════════════╝ {content} [Pipeline completed
successfully] """ print(report)

python <report import os print(open(os.environ['MSH_VAR_report']).read())
```



Pattern 2 — LLM in the Middle

What it does: Python generates structured financial data. LLM analyzes it and produces a narrative report. Python consumes the report, extracts metadata, and computes sentiment signals.

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.

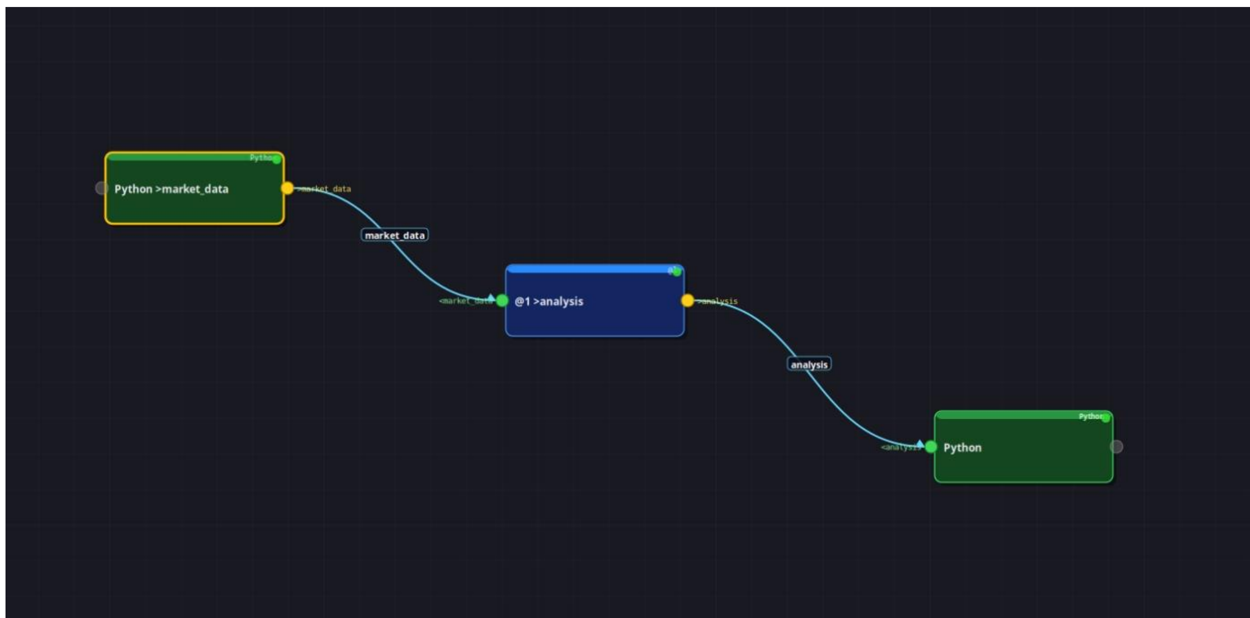
Flow diagram:

```
python >market_data
@1 <market_data >analysis   (LLM: analyst report)
python <analysis             (extract metadata + print)
```

Code:

```
python >market_data data = {     "ticker": "TECHX",     "week": ["Mon",
"Tue", "Wed", "Thu", "Fri"],     "prices": [142.5, 138.2, 145.8, 151.3,
149.7],     "volume": [2.1, 3.4, 1.8, 2.9, 2.2] } lines = [f"{d}:
price=${p:.1f}, vol={v:.1f}M" for d, p, v in zip(data['week'],
data['prices'], data['volume'])] print(f"Ticker: {data['ticker']}")
print('\n'.join(lines))
```

```
python <analysis import os text =
open(os.environ['MSH_VAR_analysis']).read().strip() words = text.split()
sentences = sum(1 for c in text if c in '.!?') positive = sum(1 for w in
words if w.lower() in
['growth','strong','up','gain','positive','rising','increased']) negative =
sum(1 for w in words if w.lower() in
['drop','weak','down','loss','negative','falling','decreased'])
print(f"[Meta] Words: {len(words)} | Sentences: {sentences} | +signals:
{positive} | -signals: {negative}") print("--- LLM Analysis ---") print(text)
```



Pattern 3 — Fan-Out (One Variable → Many Consumers)

What it does: A single Python block generates a corpus of text. Three independent consumers process it simultaneously: lexical statistics, sentence structure analysis, and LLM semantic analysis — all from the same variable.

Key concept: Multiple blocks can reference the same `<varname`. Execution is sequential in document order, but all consumers read the same unchanged file.

Flow diagram:

```
python >corpus
  ↓ ↓ ↓
python <corpus          (lexical stats)
python <corpus          (sentence structure)
@1 <corpus >semantics   (LLM semantic analysis)
python <semantics       (print)
```

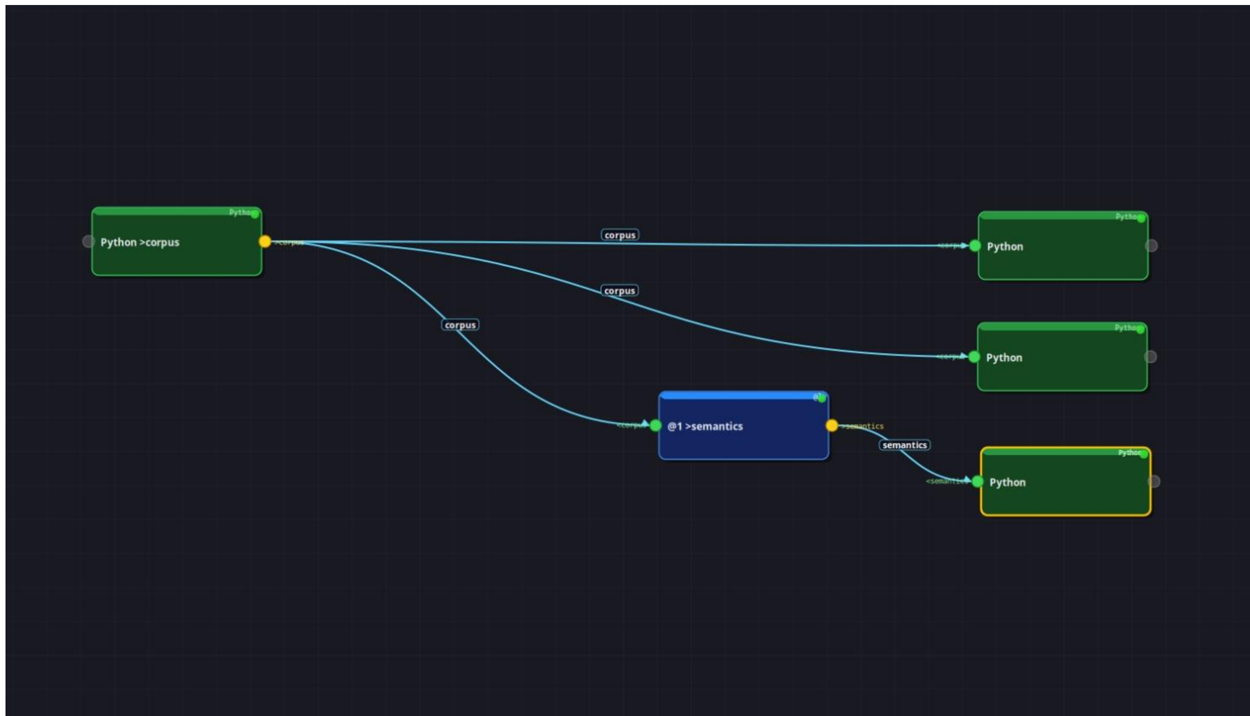
Code:

```
python >corpus text = """ Artificial intelligence is transforming every
industry. From healthcare to finance, machine learning models detect patterns
humans cannot see. Natural language processing enables computers to
understand context and nuance. Neural networks learn hierarchical features
from raw data automatically. The future belongs to those who can harness
these powerful tools responsibly. """ print(text.strip())
```

```
python <corpus import os, re text =
open(os.environ['MSH_VAR_corpus']).read().strip() words = text.split() unique
= set(w.lower().strip('.,!?') for w in words) avg_word_len = sum(len(w) for w
in words) / len(words) print(f"[Lexical] Total words: {len(words)} | Unique:
{len(unique)} | Avg word length: {avg_word_len:.1f}")
```

```
python <corpus import os text =
open(os.environ['MSH_VAR_corpus']).read().strip() sentences = [s.strip() for
s in text.split('.') if s.strip()] lengths = [len(s.split()) for s in
sentences] print(f"[Structure] Sentences: {len(sentences)} | Avg length:
{sum(lengths)/len(lengths):.1f} words | Longest: {max(lengths)} words")
```

```
python <semantics import os print("[Semantic Analysis]")
print(open(os.environ['MSH_VAR_semantics']).read().strip())
```



Pattern 4 — LLM Code Generation → Execute via Variable

What it does: Python defines a coding task. LLM generates the solution as executable Python code. A second Python block inspects, executes, and validates it — no manual copy-paste needed.

Key concept: `os.environ['MSH_VAR_code']` contains the file path. `subprocess.run([sys.executable, path])` executes the generated code as a subprocess.

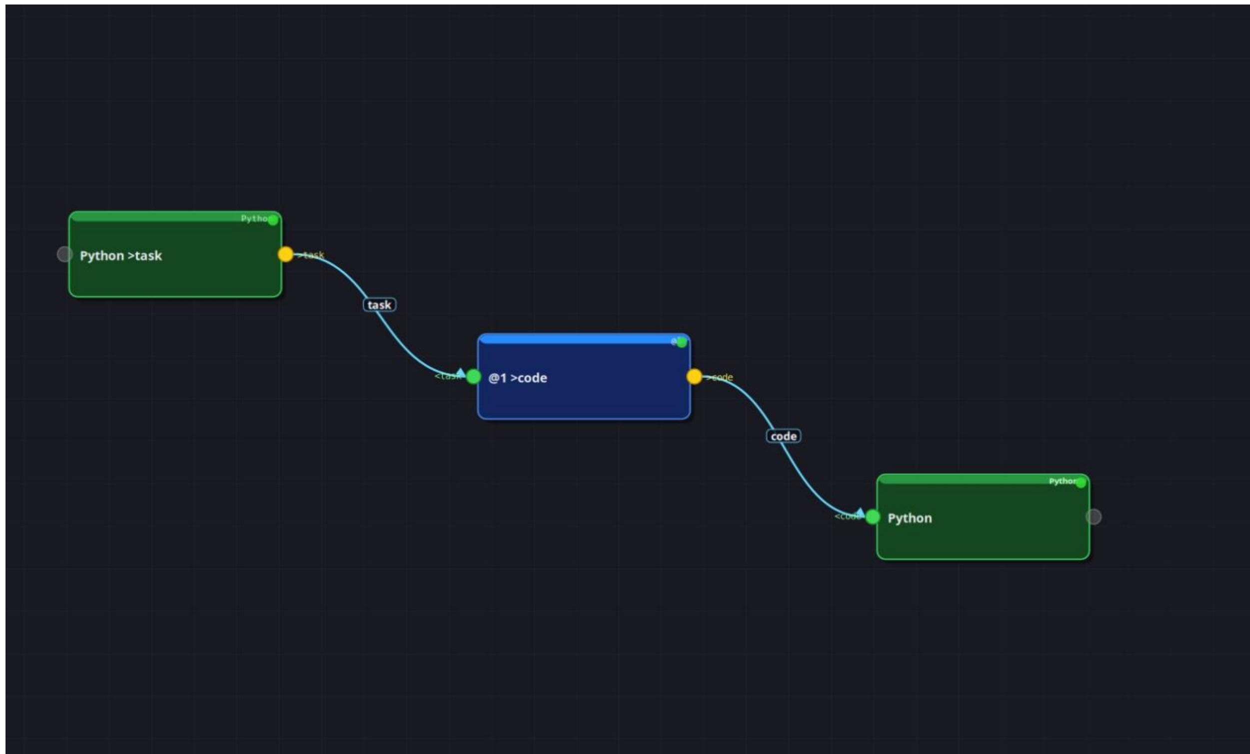
Flow diagram:

```
python >task
@1 <task >code   (LLM: generate Python code)
python <code     (inspect + execute)
```

Code:

```
python >task task = """ Write a Python function called analyze_text(text)
that counts word frequency and returns the top 5 most common words with their
counts. Call it with: 'to be or not to be that is the question' Print results
as: word: count """ print(task.strip())
```

```
python <code import os, subprocess, sys path = os.environ['MSH_VAR_code']
print("=== Generated Code ===") print(open(path).read()) print("=== Execution
Output ===") result = subprocess.run([sys.executable, path],
capture_output=True, text=True) print(result.stdout) if result.stderr:
print("STDERR:", result.stderr)
```



Pattern 5 — Two-LLM Review Chain

What it does: Python defines a data science task. Model 1 generates Python code. Model 2 reviews it for correctness and style. Model 1 receives both original code and review and produces an improved version. The final code is executed and validated.

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

Flow diagram:

```
python >task
@1 <task >code                (generate initial code)
python <code                  (print)
@2 <code >review              (review: correctness + style)
python <review                (print)
@1 <code <review >improved    (generate improved version)
python <improved              (execute final code)
```

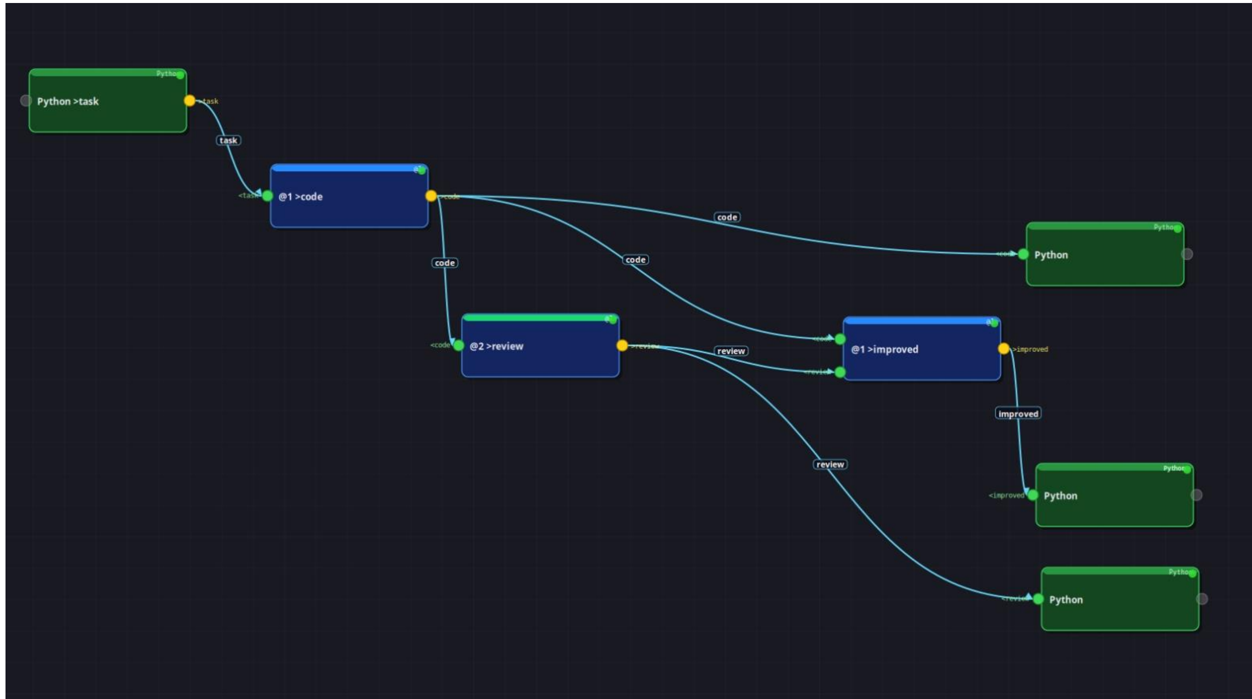
Code:

```
python >task print("Write a Python function `moving_average(data, window)`
that computes the moving average of a list. Handle edge cases: window larger
than data, empty list, window=1. Include a demo with data=[1,3,5,7,9,11,13]
and window=3.")
```

```
python <code import os print("=== Model 1: Initial Code ===")
print(open(os.environ['MSH_VAR_code']).read())
```

```
python <review import os print("=== Model 2: Code Review ===")
print(open(os.environ['MSH_VAR_review']).read())
```

```
python <improved import os, subprocess, sys path =
os.environ['MSH_VAR_improved'] print("=== Final Improved Code ===")
print(open(path).read()) print("=== Execution ===") result =
subprocess.run([sys.executable, path], capture_output=True, text=True)
print(result.stdout) if result.stderr:     print("STDERR:", result.stderr)
```



Pattern 6 — Parallel 3-Model Query

What it does: Python formulates a complex question. All three LLM models answer it independently. A Python block collects and compares all three responses with word-count statistics.

Key concept: Three LLM directive blocks with different model numbers (@1, @2, @3) all read the same `<question` variable and write to separate output variables.

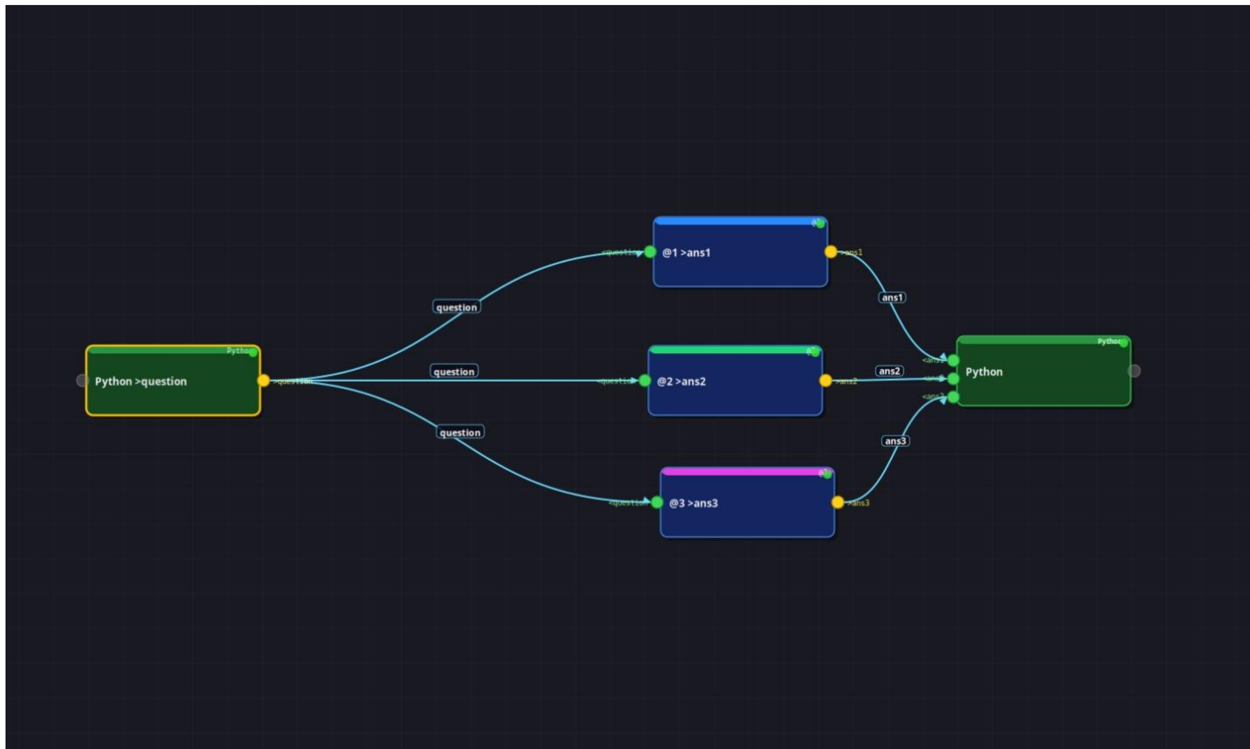
Flow diagram:

```
python >question
@1 <question >ans1
@2 <question >ans2
@3 <question >ans3
python <ans1 <ans2 <ans3  (compare + display)
```

Code:

```
python >question print("Given a dataset of 1 million rows, how would you
efficiently find all duplicate records in Python? Describe your data
structure choice, time complexity, and Python-specific optimizations. Answer
in 3-5 sentences.")
```

```
python <ans1 <ans2 <ans3 import os answers = {     'Model 1 (Technical)':
open(os.environ['MSH_VAR_ans1']).read().strip(),     'Model 2 (Libraries)':
open(os.environ['MSH_VAR_ans2']).read().strip(),     'Model 3 (Scalability)':
open(os.environ['MSH_VAR_ans3']).read().strip(), } for label, text in
answers.items():     print(f"=== {label} ({len(text.split())} words) ===")
print(text)     print()
```



Pattern 7 — Evaluator-Optimizer Loop

What it does: Python defines a validation task. Model 1 generates Python code. Model 2 evaluates it and returns `ACCEPTED` or `REJECTED` with feedback. The loop repeats until accepted or max iterations reached.

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

Flow diagram:

```
python >task
[LOOP max=3 until=verdict:ACCEPTED]
```

```
  @1 <task >code          (generate)
  python <code            (print)
  @2 <code >verdict       (evaluate: ACCEPTED or REJECTED + reason)
  python <verdict         (print)
[END_LOOP]
python <code              (execute final accepted code)
```

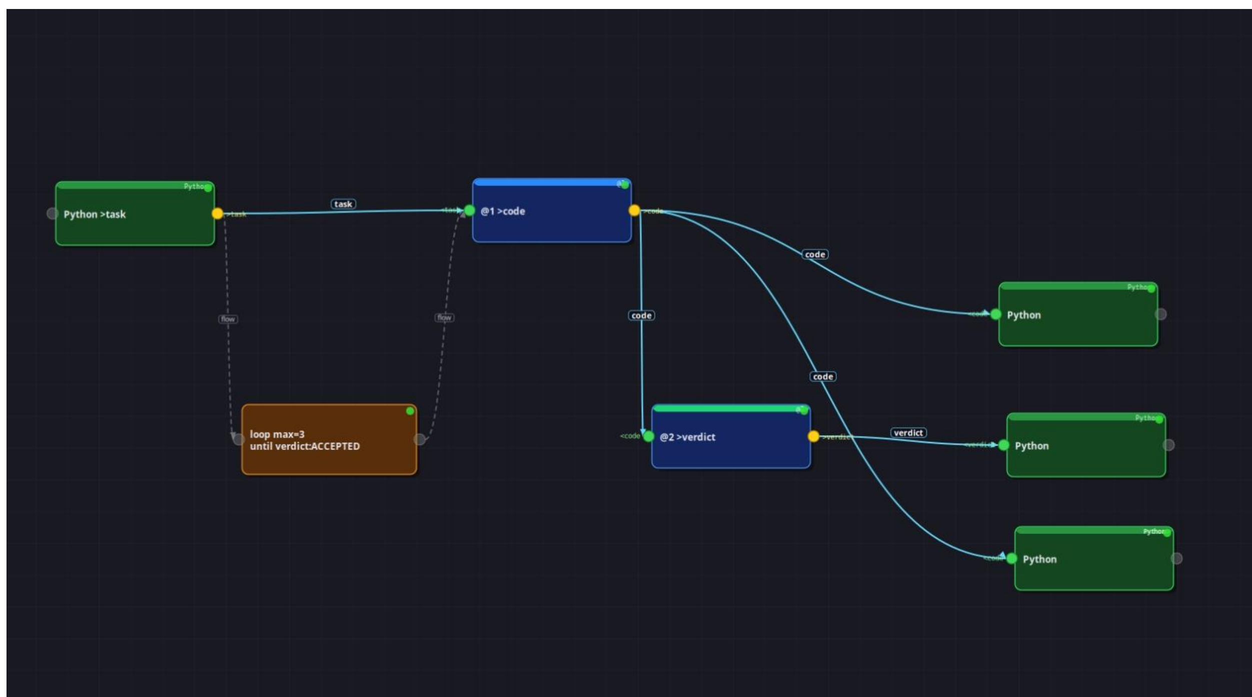
Code:

```
python >task print("Write a Python function `validate_email_list(emails)`
that takes a list of strings and returns a dict with keys 'valid' and
'invalid', each containing the respective emails. Use regex. Must handle None
and empty strings gracefully.")
```

```
python <code import os print("=== Generated Code ===")
print(open(os.environ['MSH_VAR_code']).read())
```

```
python <verdict import os print("=== Verdict ===",
open(os.environ['MSH_VAR_verdict']).read().strip())
```

```
python <code import os, subprocess, sys path = os.environ['MSH_VAR_code']
print("=== Final Accepted Code ===") print(open(path).read()) print("===
Execution ===") result = subprocess.run([sys.executable, path],
capture_output=True, text=True) print(result.stdout) if result.stderr:
print("STDERR:", result.stderr)
```



Pattern 8 — Multi-Stage Python Analysis Pipeline

What it does: Python generates a synthetic HR dataset. Python computes statistics. LLM @1 provides a narrative analysis. LLM @2 generates actionable recommendations. Python formats a complete management report.

Key concept: Language blocks and LLM blocks are interchangeable pipeline stages — any block can consume outputs from any previous block regardless of what produced them.

Flow diagram:

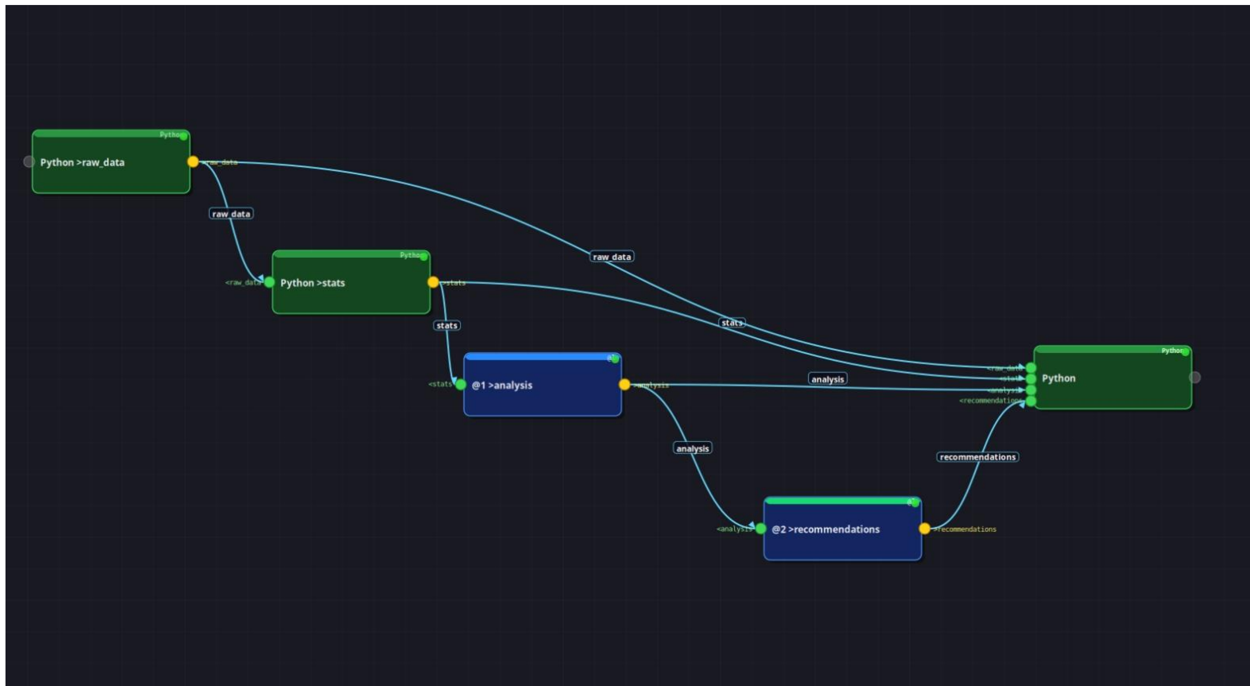
```
python >raw_data          (generate employee dataset)
python <raw_data >stats   (compute department statistics)
@1 <stats >analysis       (LLM: narrative analysis)
@2 <analysis >recommendations  (LLM: actionable advice)
python <raw_data <stats <analysis <recommendations  (final report)
```

Code:

```
python >raw_data import random random.seed(99) depts = ['Engineering',
'Marketing', 'Sales', 'HR'] print("dept,salary,tenure,perf_score") for i in
range(20):     dept = random.choice(depts)     salary = random.randint(55000,
130000)     tenure = random.randint(1, 15)     score =
round(random.uniform(2.5, 5.0), 1)
print(f"{dept},{salary},{tenure},{score}")
```

```
python <raw_data >stats import os from collections import defaultdict lines =
open(os.environ['MSH_VAR_raw_data']).read().strip().split('\n') rows =
[l.split(',') for l in lines[1:]] dept_data = defaultdict(lambda:
{'salaries': [], 'tenures': [], 'scores': []}) for dept, salary, tenure,
score in rows:     dept_data[dept]['salaries'].append(int(salary))
dept_data[dept]['tenures'].append(int(tenure))
dept_data[dept]['scores'].append(float(score)) for dept, d in
sorted(dept_data.items()):     avg_sal = sum(d['salaries']) /
len(d['salaries'])     avg_ten = sum(d['tenures']) / len(d['tenures'])
avg_scr = sum(d['scores']) / len(d['scores'])     print(f"{dept}:
n={len(d['salaries'])} avg_salary=${avg_sal:.0f} avg_tenure={avg_ten:.1f}yr
avg_score={avg_scr:.2f}")
```

```
python <raw_data <stats <analysis <recommendations import os raw =
open(os.environ['MSH_VAR_raw_data']).read().strip().split('\n') stats =
open(os.environ['MSH_VAR_stats']).read().strip() analysis =
open(os.environ['MSH_VAR_analysis']).read().strip() recs =
open(os.environ['MSH_VAR_recommendations']).read().strip() print("=" * 50)
print("       HR ANALYTICS MANAGEMENT REPORT") print("=" * 50)
print(f"\n[Dataset] {len(raw)-1} employee records\n") print("[Statistics]\n"
+ stats) print("\n[Analysis]\n" + analysis) print("\n[Recommendations]\n" +
recs) print("\n" + "=" * 50)
```



Pattern 9 — Routing (LLM Classifies → Conditional Branch Executes)

What it does: Python generates a user query. LLM classifies it into `STATISTICS`, `ALGORITHM`, or `EXPLANATION`. Subsequent Python blocks and LLM calls each carry an `if=route:VALUE` condition — only the matching block executes.

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

Flow diagram:

```
python >query
@1 <query >route   (classify: STATISTICS / ALGORITHM / EXPLANATION)
python <query if=route:STATISTICS   (run stats computation)
python <query if=route:ALGORITHM    (run algorithmic demo)
@1 <query if=route:EXPLANATION      (LLM explains in plain language)
python <route  (print classification)
```

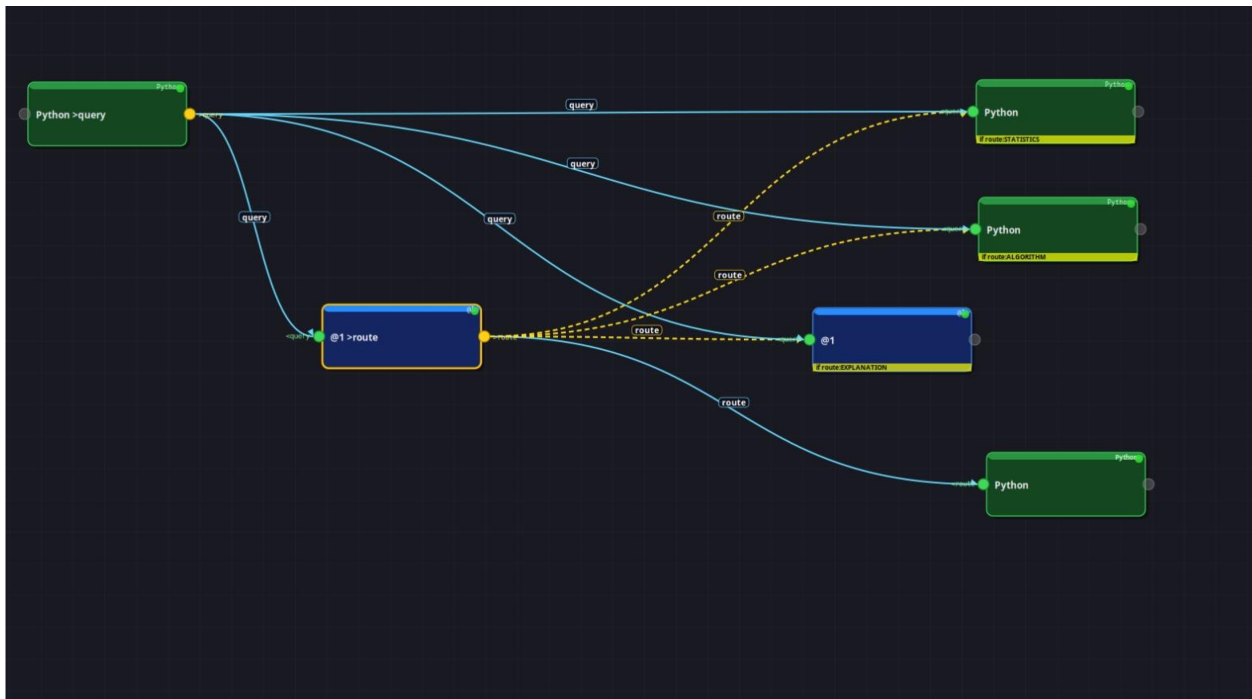
Code:

```
python >query print("Given a sorted array and a target value, what is the
most efficient way to find if the target exists?")
```

```
python <query if=route:STATISTICS import os, statistics print("=== STATISTICS
branch ===") data = [2, 4, 4, 4, 5, 5, 7, 9] print(f"Mean:
{statistics.mean(data)}") print(f"Median: {statistics.median(data)}")
print(f"Stdev: {statistics.stdev(data):.3f}")
```

```python &lt;query if=route:ALGORITHM import os print(“=== ALGORITHM branch ===”) def binary_search(arr, target): lo, hi = 0, len(arr) - 1 while lo <= hi: mid = (lo + hi) // 2 if arr[mid] == target: return mid elif arr[mid] < target: lo = mid + 1 else: hi = mid - 1 return -1

arr = list(range(0, 100, 2)) for target in [42, 57, 88]: idx = binary_search(arr, target) print(f”Search for {target}: {‘found at index’ + str(idx) if idx != -1 else ‘not found’}“) print(”Time complexity: O(log n)“)

````
<!--@1 <query if=route:EXPLANATION
Answer this question in plain, accessible language without code. 2-3 sentence
s.
-->

```python <route
import os
print(f"\nQuery classified as: {open(os.environ['MSH_VAR_route']).read().stri
p()}")
````



Pattern 10 — Full Pipeline (All Patterns Combined)

What it does: Python generates primes and statistics. Two LLM models analyze and poeticize in parallel (async). Barrier waits for both. Third LLM synthesizes. Python formats with decorative framing.

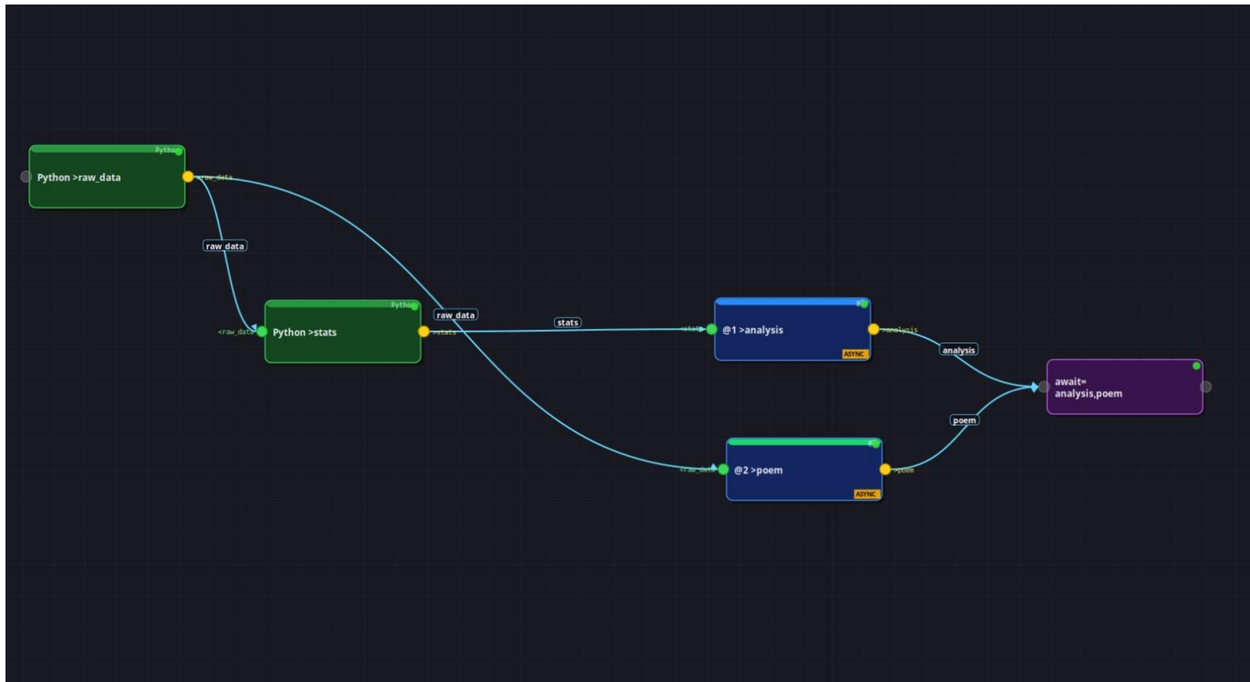
Key concept: There is no special "combine patterns" syntax — patterns compose naturally because every block simply reads from named variables. The document structure itself defines the execution graph.

Flow diagram:

```
python >raw_data          (generate primes via sieve)
python <raw_data >stats   (compute statistics)
@1 <stats >analysis  async ─┐
@2 <raw_data >poem   async ─┤
python await=analysis,poem ─┘
@1 <analysis <poem >combined  (synthesize)
python <combined              (format + display)
```

Code:

```python >raw_data def sieve(n): is_prime = [True] * (n + 1) is_prime[0] = is_prime[1] = False for i in range(2, int(n**0.5) + 1): if is_prime[i]: for j in range(i*i, n+1, i): is_prime[j] = False return [i for i in range(2, n+1) if is_prime[i]]

primes = sieve(50)[:12] print(' '.join(map(str, primes)))

````
```python <raw_data >stats
import os
nums = list(map(int, open(os.environ['MSH_VAR_raw_data']).read().strip().spli
t()))
mean = sum(nums) / len(nums)
gaps = [nums[i+1] - nums[i] for i in range(len(nums)-1)]
avg_gap = sum(gaps) / len(gaps)
print(f"First {len(nums)} primes: {nums}")
print(f"Sum={sum(nums)} Mean={mean:.2f} Avg_gap={avg_gap:.2f} Largest={max(nu
ms)}")

python await=analysis,poem

python <combined import os combined =
open(os.environ['MSH_VAR_combined']).read().strip() width = max(60,
len(combined) + 4) border = '=' * width print(f"╔{border}╗") print(f"║
{'PRIME NUMBERS — FINAL SYNTHESIS':^{width-2}}  ║") print(f"╠{border}╣") for
line in combined.split('. '):     line = line.strip()     if line:
print(f"║  {line:<{width-2}}  ║") print(f"╚{border}╝")
````



Pattern 11 — Chained LLM Nodes (Python Style)

What it does: Two Python blocks chain two LLM calls: the first generates an explanation using two input variables (topic + style), the second extracts structured metadata. Python parses results and presents a formatted output.

Key concept: Python reads variables via `os.environ['MSH_VAR_name']` and the same variable system powers both code blocks and LLM directives seamlessly.

Flow diagram:

```
python >topic
python >style
@1 <topic <style >explanation   (LLM: generate explanation)
@2 <explanation >metadata       (LLM: extract keywords + audience)
python <explanation <metadata   (format + display)
```

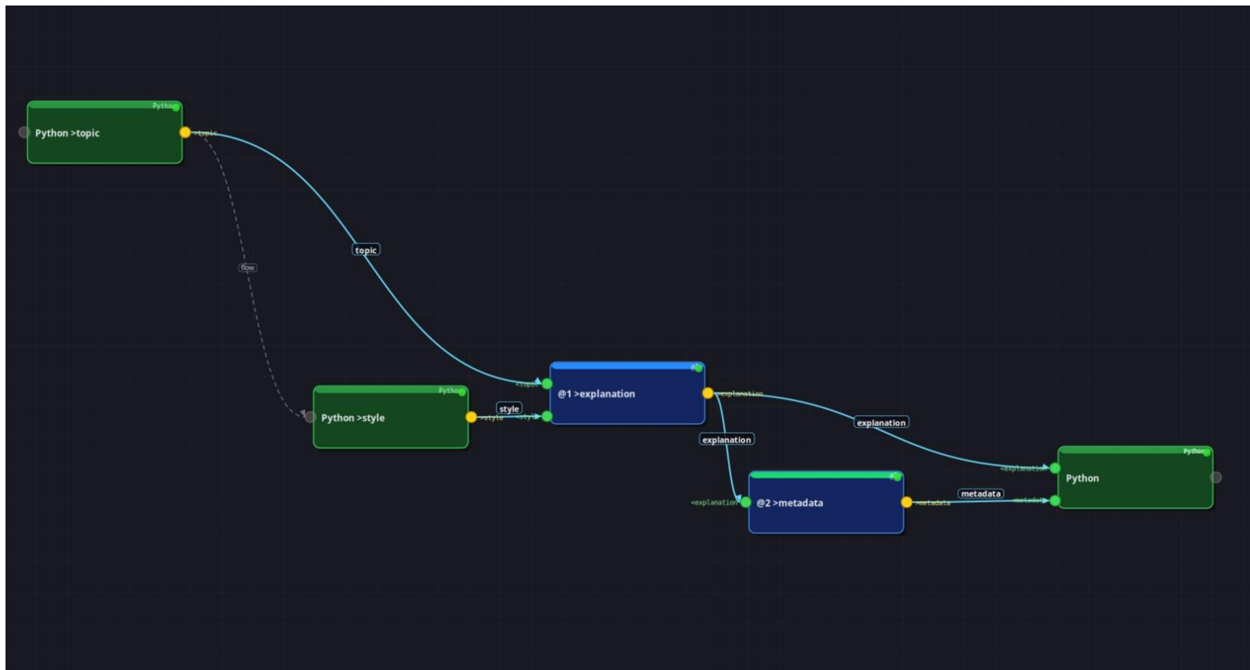
Code:

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

```
python >style print("engaging and accessible for a non-technical audience")
```

```
python <explanation <metadata import os, re explanation =
open(os.environ['MSH_VAR_explanation']).read().strip() metadata =
open(os.environ['MSH_VAR_metadata']).read().strip() kw =
re.search(r'KEYWORDS:\s*(.+)', metadata) aud = re.search(r'AUDIENCE:\s*(.+)',
metadata) read = re.search(r'READABILITY:\s*(.+)', metadata) print("===
Explanation ===") print(explanation) print("\n=== Metadata ===")
```

```
print(f"Keywords   : {kw.group(1) if kw else 'n/a'}") print(f"Audience   : {aud.group(1) if aud else 'n/a'}") print(f"Readability: {read.group(1) if read else 'n/a'}")
```



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

What it does: Three LLM models review the same Python code snippet simultaneously from three angles (performance, security, style). Python await barrier synchronizes. Fourth LLM synthesizes into one definitive assessment. Python presents the complete review report.

Key concept: `async` on an LLM directive launches it in a background process. `` `python await=...` `` blocks until all named variables are written.

Flow diagram:

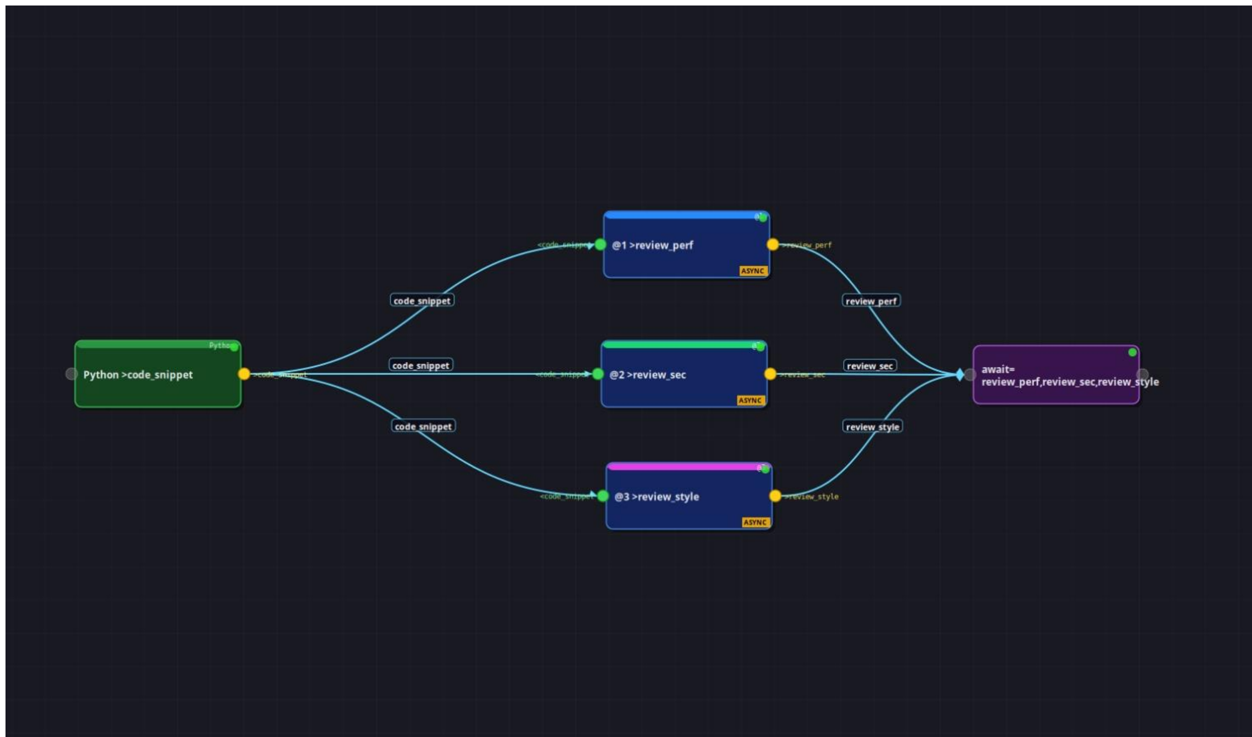
```
python >code_snippet
@1 <code_snippet >review_perf   async ┐
@2 <code_snippet >review_sec    async ┤
@3 <code_snippet >review_style  async ┘
python await=review_perf,review_sec,review_style
@1 <review_perf <review_sec <review_style >final_review
python <final_review  (format complete report)
```

Code:

```python >code_snippet code = ''' def find_user(users_list, user_id): for user in users_list: if user["id"] == user_id: return user["password"] return None
```

users = [{“id”: i, “password”: f”pass{i}“} for i in range(10000)] result = find_user(users, 9999) print(result) ’’’ print(code.strip())

~~~
<!--@1 <code_snippet >review_perf async
Review this Python code focusing ONLY on performance and time complexity. One paragraph with Big-O analysis and a concrete optimization suggestion.
-->

<!--@2 <code_snippet >review_sec async
Review this Python code focusing ONLY on security vulnerabilities. One paragraph identifying specific risks and suggested fixes.
-->

<!--@3 <code_snippet >review_style async
Review this Python code focusing ONLY on Python style, PEP 8, type hints, and naming conventions. One paragraph.
-->

```python await=review_perf,review_sec,review_style

python <final_review import os review =
open(os.environ['MSH_VAR_final_review']).read().strip()
print("╔══════════════════════════════════════╗") print("║      ENSEMBLE CODE REVIEW REPORT      ║") print("╚══════════════════════════════════════╝")
print(review) print("\n[Generated from 3 independent model perspectives]")
~~~



Part II — Advanced Node Types (13–24)

Pattern 13 — WHILE Loop: Iterative Convergence with LLM Commentary

What it does: Computes Fibonacci numbers iteratively. Each iteration Python increments the counter, computes the next Fibonacci value, and updates loop control variables directly via `$MSH_VAR_*` file writes. LLM generates one interesting mathematical fact about each Fibonacci number. Loop exits after 5 iterations.

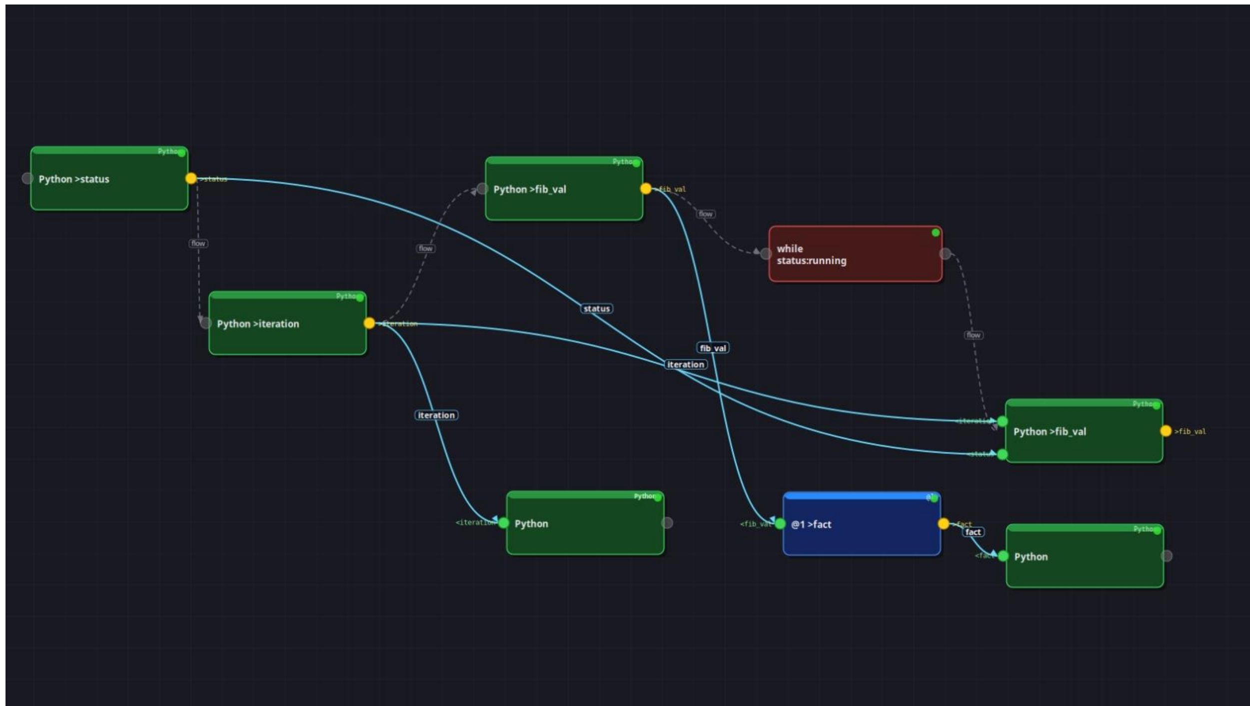
Key patterns: - Python block with multiple `>outvar` writes to each file directly via `open(os.environ['MSH_VAR_*'], 'w')`. - WHILE reads the last non-empty line of the condition variable. - Use a `running`/`done` flag as the exit condition.

Flow diagram:

```
python >status="running", >iteration="0", >fib_val="1"
[WHILE status:running]
  python <iteration <status >iteration >status >fib_val  (compute Fibonacci; write via MSH_VAR_*)
  @1 <fib_val >fact   (LLM: interesting math fact)
  python <fact        (print)
[END_WHILE]
python <iteration  (final output)
```

Code:

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

python >iteration print("0")

python >fib_val print("1")

python <iteration <status >iteration >status >fib_val import os val = int(open(os.environ['MSH_VAR_iteration']).read().strip()) val += 1 a, b = 1, 1 for _ in range(val - 1):     a, b = b, a + b fib = a with open(os.environ['MSH_VAR_iteration'], 'w') as f: f.write(str(val)) with open(os.environ['MSH_VAR_fib_val'], 'w') as f: f.write(str(fib)) with open(os.environ['MSH_VAR_status'], 'w') as f: f.write("done" if val >= 5 else "running")

python <fact import os itr = open(os.environ['MSH_VAR_iteration']).read().strip() fib = open(os.environ['MSH_VAR_fib_val']).read().strip() fact = open(os.environ['MSH_VAR_fact']).read().strip() print(f"[iter {itr}] Fib={fib} → {fact}")

python <iteration import os print(f"=== WHILE done. Completed {open(os.environ['MSH_VAR_iteration']).read().strip()} iterations. ===")
```



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

What it does: Python creates a newline-separated list of 5 Python libraries. FOREACH iterates line by line. LLM describes each library and its primary use case. Python prints formatted results. After all iterations a summary block runs.

Key patterns: - `print("a\nb\nc")` is the correct way to create a list for FOREACH. - The iterator variable is set automatically by the runtime before each iteration.

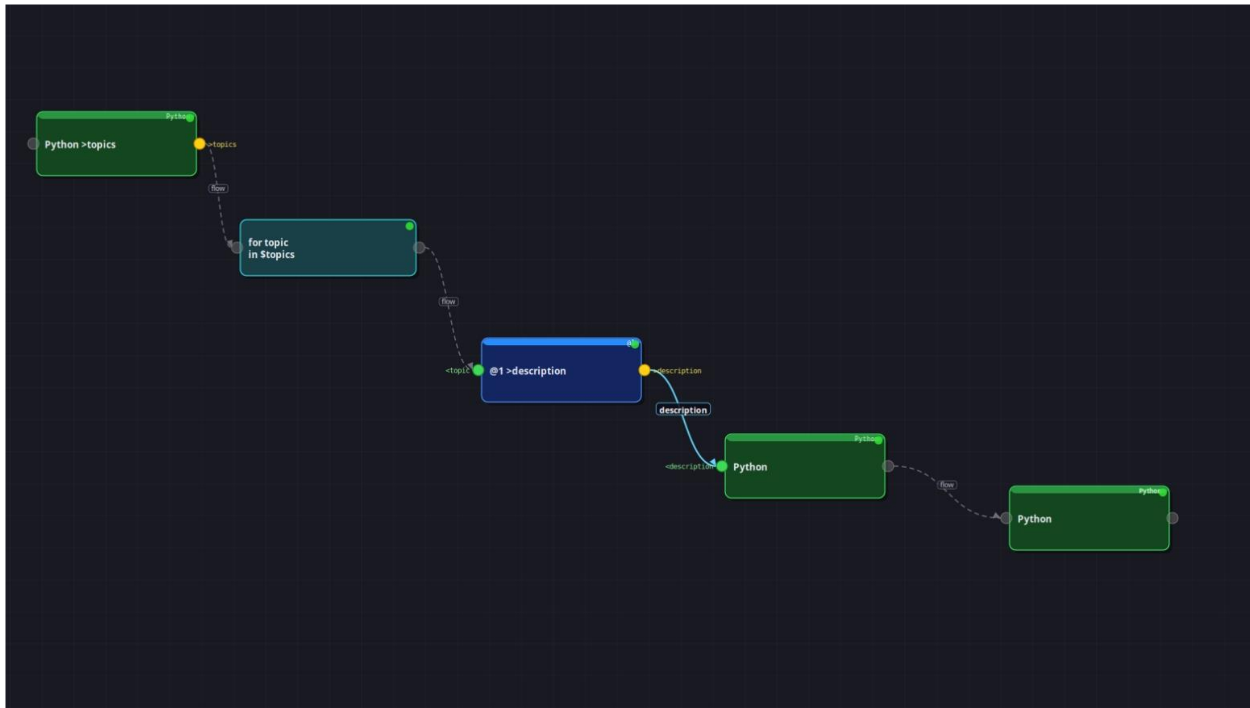
Flow diagram:

```
python >topics = "pandas\nnumpy\nscikit-learn\nmatplotlib\nfastapi"
[FOREACH topic in topics]
  @1 <topic >description  (LLM: describe the library)
  python <description     (print)
[END_FOREACH]
python  (completion summary)
```

Code:

```
python >topics print("pandas\nnumpy\nscikit-learn\nmatplotlib\nfastapi")

python <description import os topic =
open(os.environ['MSH_VAR_topic']).read().strip() desc =
open(os.environ['MSH_VAR_description']).read().strip() print(f"▶ {topic}")
print(f"  {desc}") print()
```

```
print("=== All 5 Python libraries described. ===")
```



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

What it does: Python block initializes a malformed JSON string. TRY block attempts `json.loads` — fails and triggers CATCH. CATCH prints a hardcoded message. Safe fallback uses regex extraction. Final Python block prints the recovered data.

Key patterns: - `|| exit 1` guarantees non-zero exit code on any Python error. - CATCH block does not use <errvar — prints "try_block_failed" directly as a literal. - Pipeline continues normally after <!--@end_try-->.

Flow diagram:

```
python >raw_json = '{"name": "Alice", "city": broken}'
[TRY]
  python <raw_json >parsed   (json.loads — will fail)
[CATCH >error]
  python  (print literal "try_block_failed" message)
[END_TRY]
python <raw_json >safe_result  (safe regex fallback)
python <safe_result            (print)
```

Code:

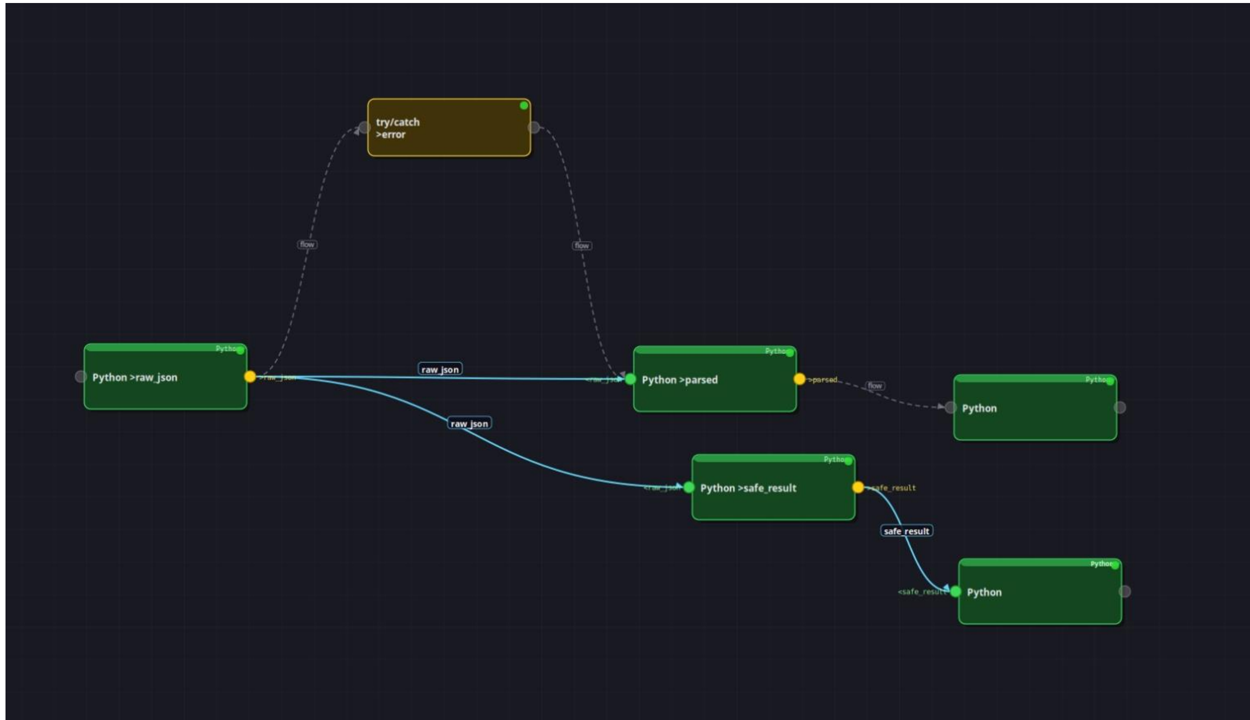
```
python >raw_json print('{"name": "Alice", "age": 30, "city": broken_value}')
```

```
python <raw_json >parsed import os, json raw =
open(os.environ['MSH_VAR_raw_json']).read().strip() data = json.loads(raw)
print(data)
```

```
print("=== Caught error: try_block_failed ===")
print("Malformed JSON detected. Switching to safe fallback parser.")
```

```
python <raw_json >safe_result import os, re raw =
open(os.environ['MSH_VAR_raw_json']).read().strip() fields =
re.findall(r'"(\w+)":\s*(?:"([^"]*)"|\d+|(\w+))', raw) result = {k: v1 or v2
for k, v1, v2 in fields} print(f"Safe extraction: {result}")
```

```
python <safe_result import os print("=== Safe Result ===")
print(open(os.environ['MSH_VAR_safe_result']).read().strip())
```



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

What it does: Python creates a two-series climate dataset. SPLIT divides it into two variables. Two async LLM calls analyze each series in parallel. Python await synchronizes. MERGE marks the reduce point. LLM synthesizes both analyses into a unified scientific insight.

Key patterns: - SPLIT creates variables line by line: line 1 → var_1, line 2 → var_2. - SPLIT and MERGE are visual markers — they execute no code. - async + AWAIT: total time = time of the slowest call.

Flow diagram:

```
python >dataset = "series1\nseries2"
[SPLIT dataset into 2]  → dataset_1, dataset_2
  @1 <dataset_1 >analysis1  async ─┐
```

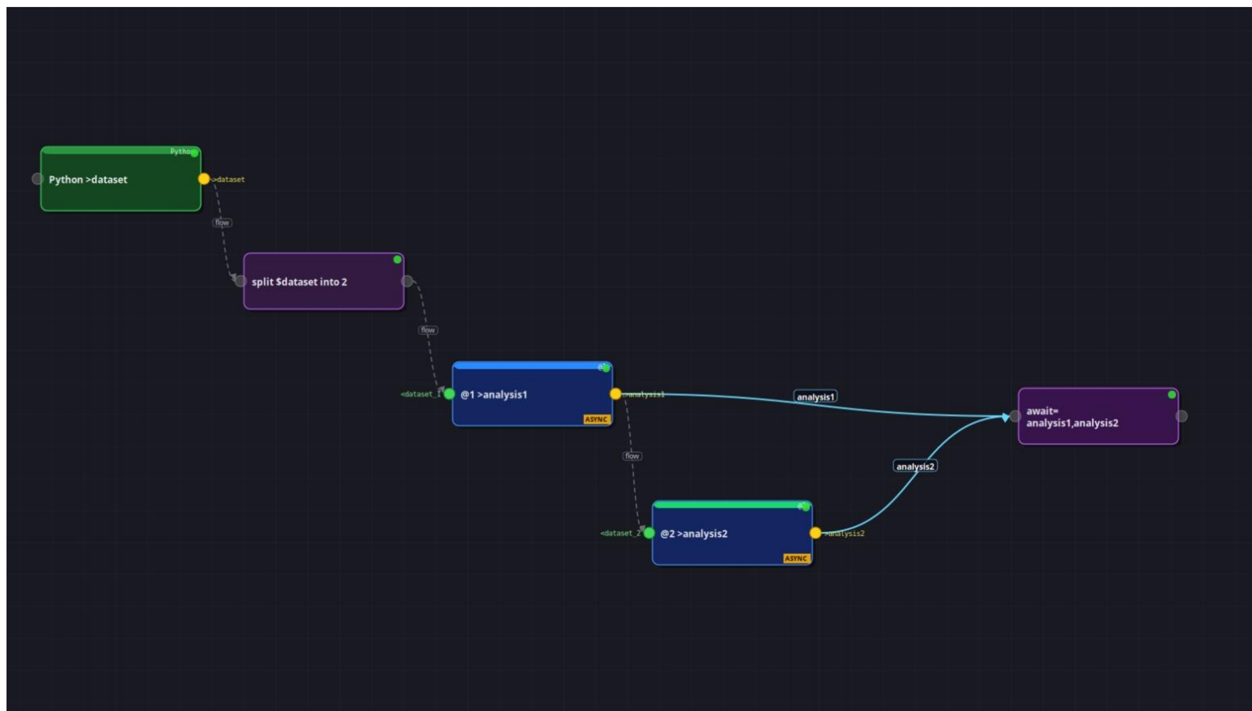
```
  @2 <dataset_2 >analysis2  async ─┤
  python await=analysis1,analysis2 ┘
[MERGE]
  @1 <analysis1 <analysis2 >combined
python <combined
```

Code:

```
python >dataset
print("climate:temp_anomaly:+1.2,+0.8,+1.5,+2.1,+1.9\nclimate:co2_ppm:315,325
,340,370,415")
```

```
python await=analysis1,analysis2
```

```
python <combined import os d1 =
open(os.environ['MSH_VAR_dataset_1']).read().strip() d2 =
open(os.environ['MSH_VAR_dataset_2']).read().strip() a1 =
open(os.environ['MSH_VAR_analysis1']).read().strip() a2 =
open(os.environ['MSH_VAR_analysis2']).read().strip() combined =
open(os.environ['MSH_VAR_combined']).read().strip() print(f"=== Series 1:
{d1} ===") print(f"Analysis: {a1}\n") print(f"=== Series 2: {d2} ===")
print(f"Analysis: {a2}\n") print(f"=== Merged Insight ===") print(combined)
```



Pattern 17 — CONFIG Node: Parameterized Pipeline

What it does: CONFIG documents pipeline parameters. Python blocks initialize runtime variables. LLM @1 generates a structured explanation. LLM @2 extracts keywords. Python assembles a formatted report. CONFIG makes the pipeline a reusable template.

Key patterns: - CONFIG is a documentation block — it does not inject variables at runtime. - Always pair CONFIG with Python `print` blocks for actual runtime values.

Flow diagram:

```
config  (domain, complexity — documentation only)
python >domain = "distributed systems"
python >complexity = "intermediate"
@1 <domain <complexity >explanation
@2 <explanation >keywords
python <explanation <keywords >report
```

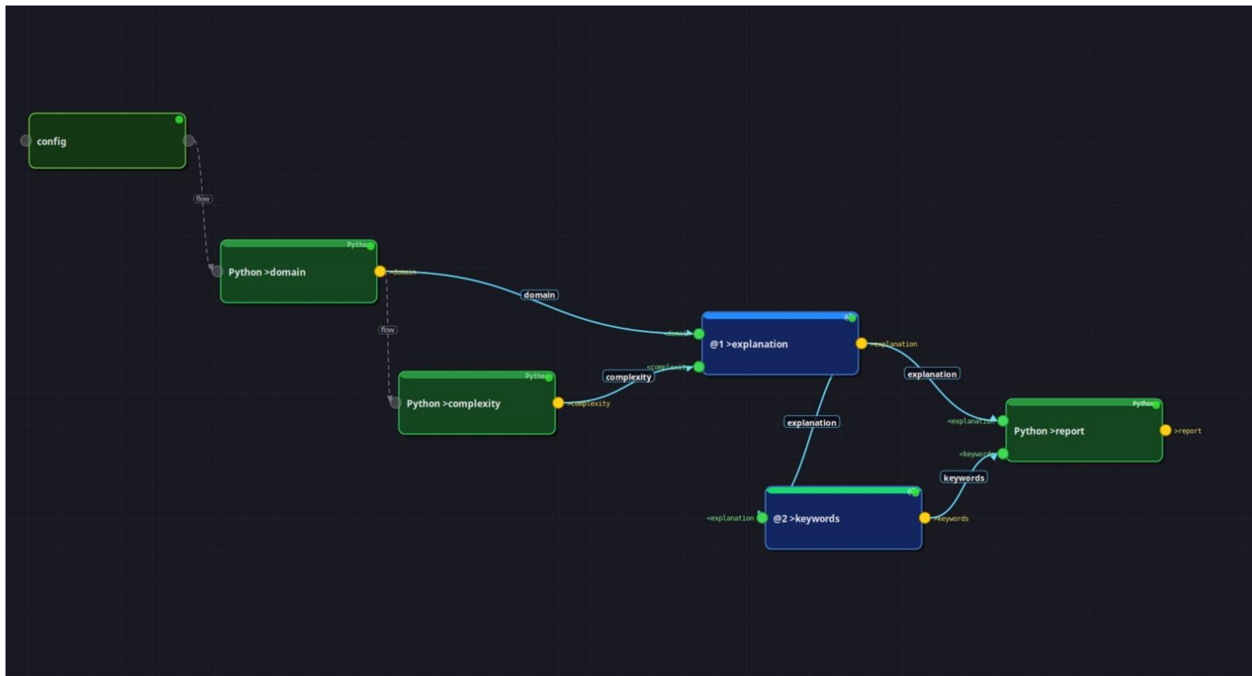
Code:

```
domain=distributed systems
complexity=intermediate
output_format=structured explanation + keywords
```

```
python >domain print("distributed systems")
```

```
python >complexity print("intermediate")
```

```
python <explanation <keywords >report import os domain =
open(os.environ['MSH_VAR_domain']).read().strip() complexity =
open(os.environ['MSH_VAR_complexity']).read().strip() explanation =
open(os.environ['MSH_VAR_explanation']).read().strip() keywords =
open(os.environ['MSH_VAR_keywords']).read().strip() print('\n'.join([
"=== Workflow Report ===",     f"Domain     : {domain}",     f"Level      :
{complexity}",     "",     "Explanation:",     explanation,     "",
f"Keywords   : {keywords}", ]))
```



Pattern 18 — FOREACH + Async LLM: Parallel Batch Processing

What it does: Python creates a list of 3 Python design patterns. FOREACH iterates. Per item: two async LLMs generate explanation and code example simultaneously. Python await synchronizes. Python prints the pair. Repeats for all items.

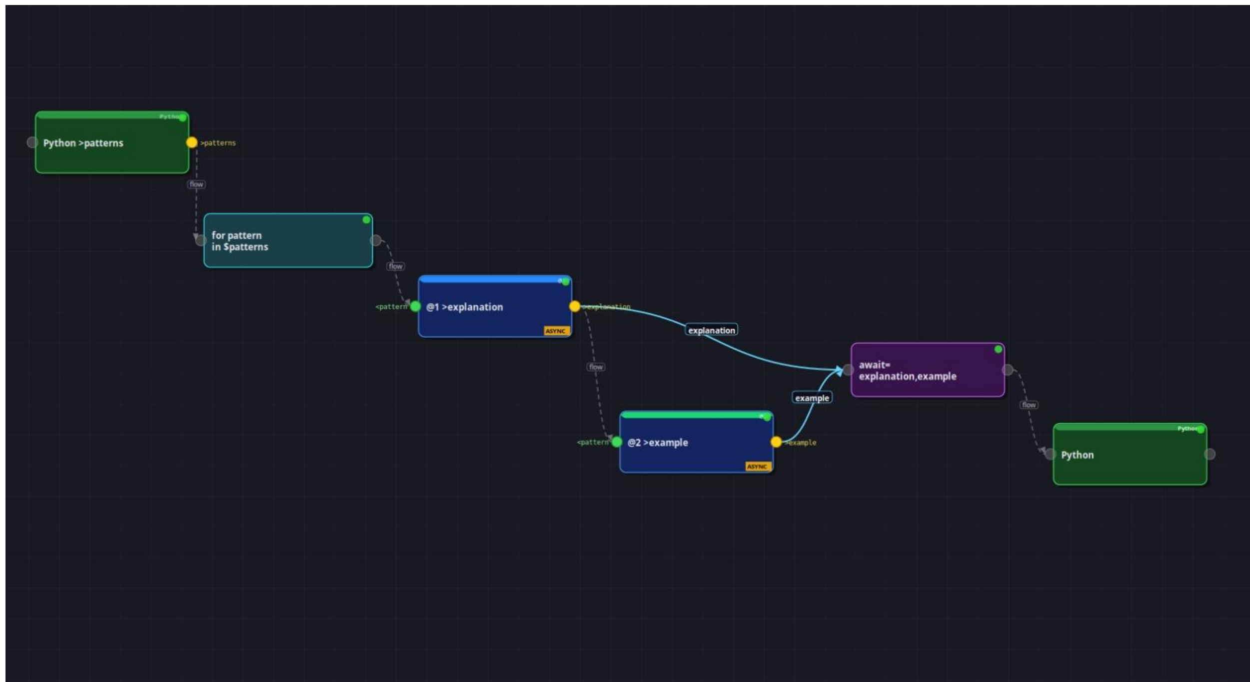
Key patterns: - `async` inside FOREACH: both LLMs run simultaneously — total time per item = slowest model. - Each async call must have exactly one `>outvar`. - The `await=` barrier lists all async output variables, comma-separated.

Flow diagram:

```
python >patterns = "singleton\ndecorator\nobserver"
[FOREACH pattern in patterns]
  @1 <pattern >explanation  async ─┐
  @2 <pattern >example      async ─┤
  python await=explanation,example ┘
  python <explanation <example  (print pair)
[END_FOREACH]
python  (completion message)
```

Code:

```
python >patterns print("singleton\ndecorator\nobserver")

python await=explanation,example

python <explanation <example import os pattern =
open(os.environ['MSH_VAR_pattern']).read().strip() exp =
open(os.environ['MSH_VAR_explanation']).read().strip() ex =
open(os.environ['MSH_VAR_example']).read().strip() print(f"===
{pattern.upper()} ===") print(f"What it is : {exp}") print(f"Python use :
{ex}") print()

print("=== All patterns processed. ===")
```



Pattern 19 — WHILE Quality Gate: Generate Until Threshold

What it does: Generates Python code for a task in a loop. LLM @1 generates a Python function. LLM @2 rates it 1–10. Python checks the score: if >= 8 writes done, otherwise `running`. Loop exits when an acceptable solution is produced.

Key patterns: - Scorer (@2) must return a bare integer — use `Reply with ONLY the integer`. - `.strip()` cleans the LLM response before numeric comparison. - All variables must be initialized before the loop.

Flow diagram:

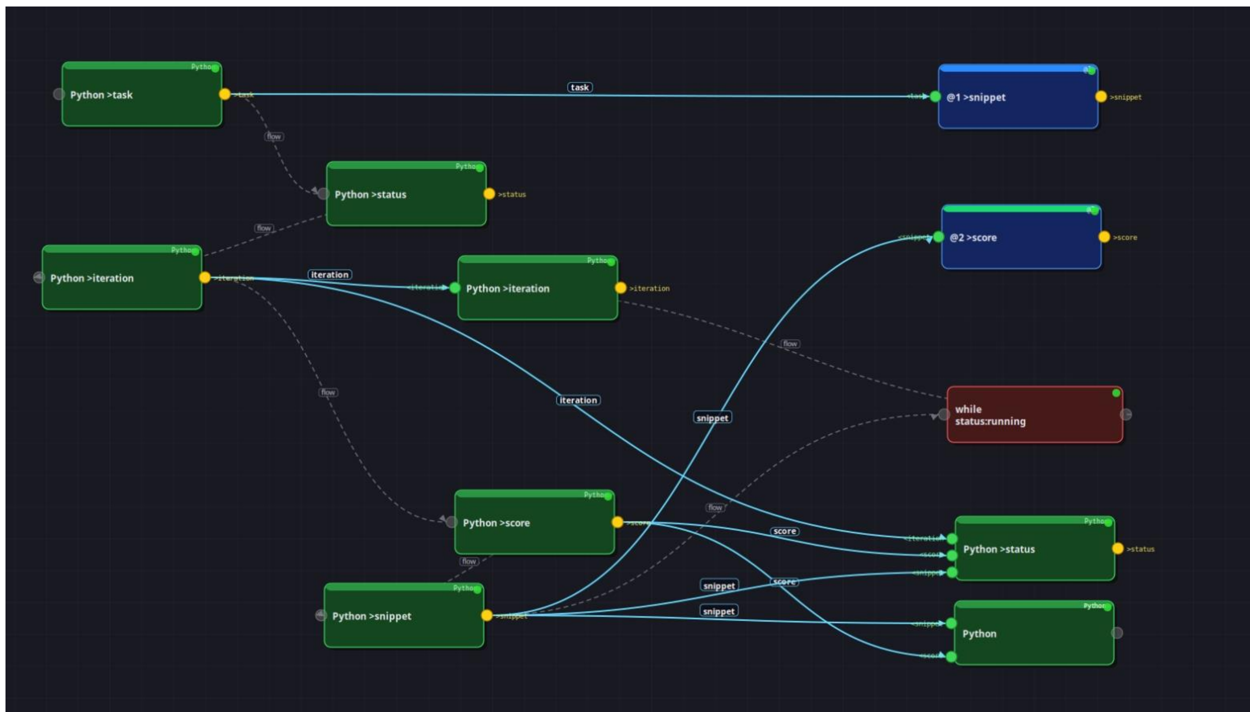
```
python >task, >status="running", >iteration="0", >score="0", >snippet=""
[WHILE status:running]
  python <iteration >iteration          (counter++)
  @1 <task >snippet                     (generate Python function)
  @2 <snippet >score                    (rate 1-10)
  python <iteration <score <snippet >status  (log + check threshold)
[END_WHILE]
python <snippet <score  (print accepted snippet)
```

Code:

```
python >task print("Write a Python function that reads a CSV file and returns
the top N rows sorted by a specified column, handling missing values
gracefully.")
```

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

```
python >iteration print("0")

python >score print("0")

python >snippet print("")

python <iteration >iteration import os val =
int(open(os.environ['MSH_VAR_iteration']).read().strip()) print(val + 1)

python <iteration <score <snippet >status import os itr =
open(os.environ['MSH_VAR_iteration']).read().strip() sc =
open(os.environ['MSH_VAR_score']).read().strip() print(f"[Iter {itr}]
Score={sc}") try:     print("done" if int(sc) >= 8 else "running") except
ValueError:     print("running")

python <snippet <score import os print(f"=== Accepted snippet
(score={open(os.environ['MSH_VAR_score']).read().strip()}) ===")
print(open(os.environ['MSH_VAR_snippet']).read().strip())
```



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

What it does: Python stores a multi-section Python documentation summary. Three Python blocks split it into chunks. Three async LLM @1 calls extract the main concept from each chunk. Python await synchronizes all three. MERGE marks the reduce point. LLM @2 synthesizes three concept labels into one theme.

Key patterns: - Map phase: N parallel LLMs, each on its own chunk — total time = slowest chunk. - Reduce phase: one LLM receives all results via multiple `<invar` with `[varname]:` labels. - AWAIT before MERGE is mandatory.

Flow diagram:

```
python >raw_text
python <raw_text >chunk1 / >chunk2 / >chunk3  (split)
  @1 <chunk1 >concept1  async ─┐
  @1 <chunk2 >concept2  async ─┤  MAP
  @1 <chunk3 >concept3  async ─┘
  python await=concept1,concept2,concept3
[MERGE]
  @2 <concept1 <concept2 <concept3 >theme  (REDUCE)
python <concept1 <concept2 <concept3 <theme  (print)
```

Code:

```
python >raw_text text = (     "Python's data model allows objects to
implement special methods that define behavior for built-in operations. "
"By implementing __len__, __getitem__, and __iter__, a class can behave like
a sequence. "     "Generators and coroutines use yield to produce lazy
sequences and enable cooperative multitasking. "     "The asyncio framework
builds on coroutines to provide concurrent I/O-bound execution on a single
thread. "     "Context managers implement __enter__ and __exit__ to guarantee
resource cleanup with the with statement." ) print(text)
```

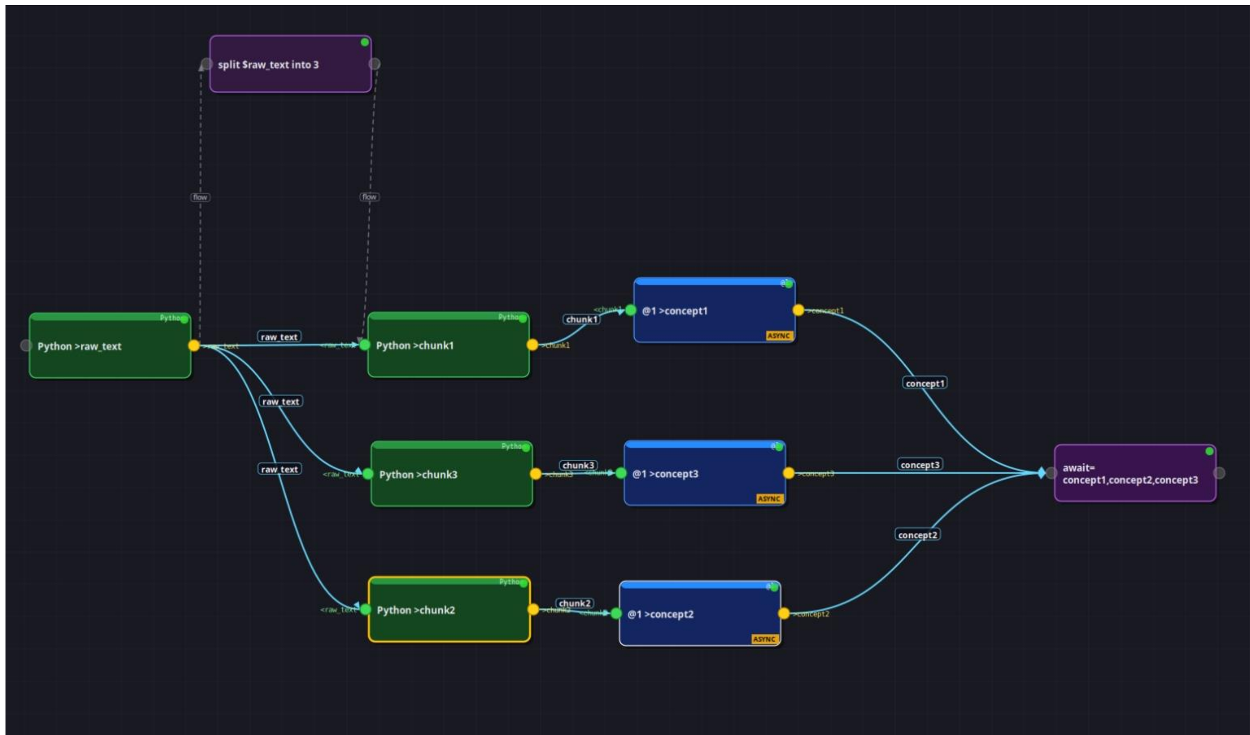
```
python <raw_text >chunk1 import os sents =
open(os.environ['MSH_VAR_raw_text']).read().strip().split('. ') sents =
[s.strip() for s in sents if s.strip()] print(sents[0] if sents else '')
```

```
python <raw_text >chunk2 import os sents =
open(os.environ['MSH_VAR_raw_text']).read().strip().split('. ') sents =
[s.strip() for s in sents if s.strip()] print('. '.join(sents[1:3]) if
len(sents) > 1 else '')
```

```
python <raw_text >chunk3 import os sents =
open(os.environ['MSH_VAR_raw_text']).read().strip().split('. ') sents =
[s.strip() for s in sents if s.strip()] print('. '.join(sents[3:]) if
len(sents) > 3 else '')
```

```
python await=concept1,concept2,concept3
```

```
python <concept1 <concept2 <concept3 <theme import os print("=== Map ===")
print(f"Chunk 1: {open(os.environ['MSH_VAR_concept1']).read().strip()}")
print(f"Chunk 2: {open(os.environ['MSH_VAR_concept2']).read().strip()}")
print(f"Chunk 3: {open(os.environ['MSH_VAR_concept3']).read().strip()}")
print("\n=== Reduce ===")
print(open(os.environ['MSH_VAR_theme']).read().strip())
```



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

What it does: LLM generates Python code for a math task. TRY executes it via subprocess. On success writes ok. On failure CATCH writes `fail`. Next iteration LLM sees the previous error and self-corrects. Loop exits when `result == ok`.

Key patterns: - CATCH block does not use `<last_error` — writes `fail` to `>result` directly. - `result` is initialized to `"fail"` before the loop. - CATCH declares `>last_error` so the parser registers the variable.

Flow diagram:

```
python >task, >result="fail", >last_error="none"
[LOOP max=3 until=result:ok]
  @1 <task <last_error >code        (generate / fix code)
  python <code                      (print)
  [TRY]
    python <code >result            (execute; writes "ok" on success)
  [CATCH >last_error]
    python >result                  (writes "fail")
  [END_TRY]
[END_LOOP]
python <result  (final status)
```

Code:

```
python >task print('Write Python code that loads JSON \'{"scores": [95, 87,
92, 78, 88]}\', computes mean and standard deviation without external
libraries, and prints: mean=XX.X std=X.X. Use only the math module.')
```

```
python >result print("fail")
```

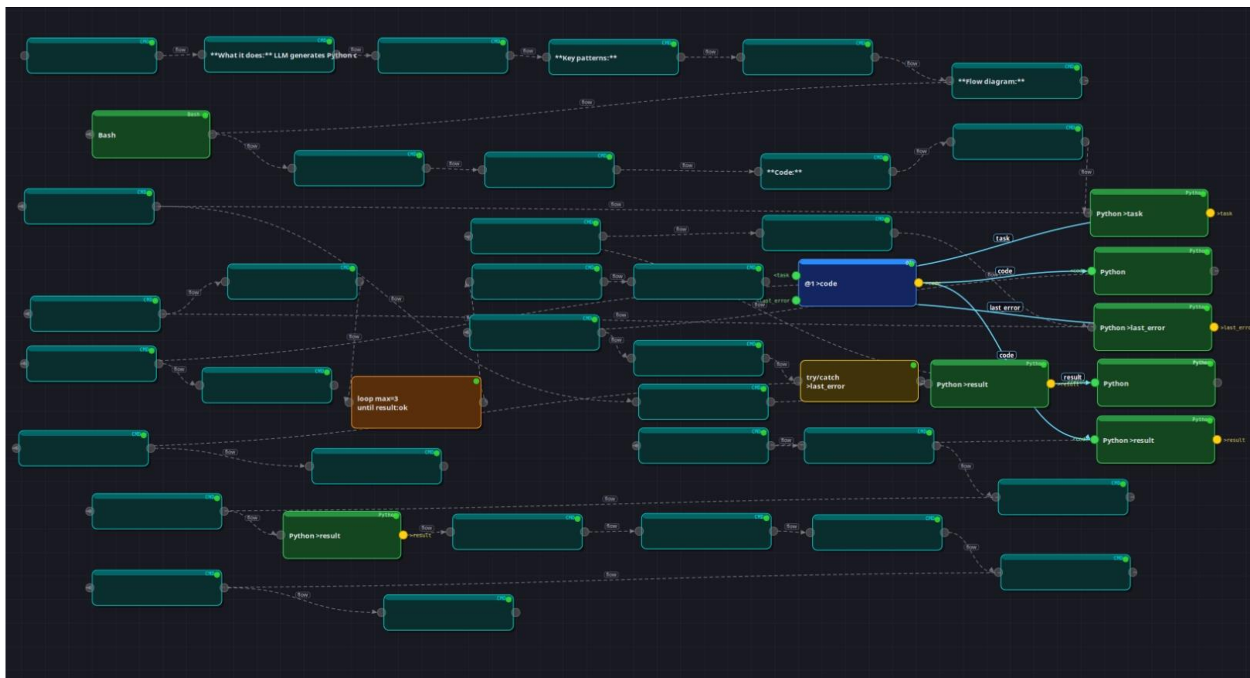
```
python >last_error print("none")
```

```
python <code import os print("=== Generated Code ===")
print(open(os.environ['MSH_VAR_code']).read())
```

```
python <code >result import os, subprocess, sys path =
os.environ['MSH_VAR_code'] result = subprocess.run([sys.executable, path],
capture_output=True, text=True) if result.returncode != 0:     raise
RuntimeError(result.stderr) print(result.stdout.strip()) print("ok")
```

```
python >result print("fail")
```

```
python <result import os print(f"=== Final status:
{open(os.environ['MSH_VAR_result']).read().strip()} ===")
```



Pattern 22 — Multi-Variable Output: Structured Field Extraction

What it does: LLM responds in a strict 3-line format to a Python library description. Python parses the response and writes three structured fields to separate variables (`summary`, `tags`, `difficulty`) directly via `MSH_VAR_*` file writes. LLM @2 generates a learning recommendation. Python prints the complete output.

Key patterns: - Multiple `>outvar` on a CODE block: block runs without stdout capture — must write via `open(os.environ['MSH_VAR_*'], 'w')`. - Multiple `>outvar` on an LLM directive: full response is copied identically into each variable. - The `MSH_VAR_*` environment variables are pre-set by the parser.

Flow diagram:

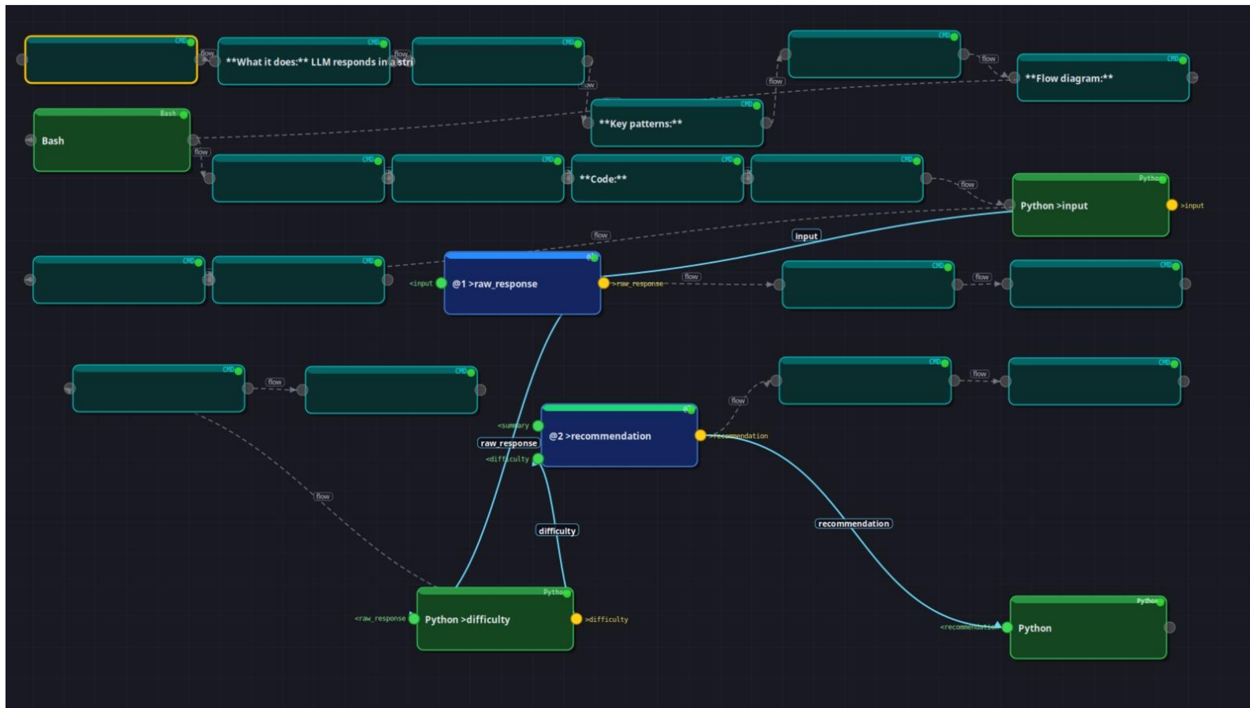
```
python >input
@1 <input >raw_response                  (LLM: structured 3-line format)
python <raw_response >summary >tags >difficulty
  (parse with regex; write via open(os.environ['MSH_VAR_*'], 'w'))
@2 <summary <difficulty >recommendation
python <recommendation  (print)
```

Code:

```
python >input print("PyTorch is an open-source machine learning framework developed by Meta AI Research. It provides dynamic computational graphs, automatic differentiation, and seamless GPU acceleration. PyTorch is widely used in research for flexibility and in production for TorchScript and TorchServe capabilities.")
```

```
python <raw_response >summary >tags >difficulty import os, re text = open(os.environ['MSH_VAR_raw_response']).read() summary = re.search(r'SUMMARY:\s*(.+)', text) tags = re.search(r'TAGS:\s*(.+)', text) difficulty = re.search(r'DIFFICULTY:\s*(.+)', text) summary = summary.group(1).strip() if summary else "n/a" tags = tags.group(1).strip() if tags else "n/a" difficulty = difficulty.group(1).strip() if difficulty else "n/a" with open(os.environ['MSH_VAR_summary'], 'w') as f: f.write(summary) with open(os.environ['MSH_VAR_tags'], 'w') as f: f.write(tags) with open(os.environ['MSH_VAR_difficulty'], 'w') as f: f.write(difficulty) print(f"Summary   : {summary}") print(f"Tags      : {tags}") print(f"Difficulty: {difficulty}")
```

```
python <recommendation import os print("\n=== Learning Recommendation ===") print(open(os.environ['MSH_VAR_recommendation']).read().strip())
```



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

What it does: CONFIG documents pipeline parameters. WHILE loop runs until explanation quality >= threshold. LLM @1 generates a 3-sentence explanation with code. LLM @2 scores it 1–10. Python checks threshold. After loop LLM @3 polishes for publication. Python prints final result with metrics.

Key patterns: - @2 receives `<target_audience` explicitly — scorer needs context. - Three distinct model roles: generator (@1), scorer (@2), finisher (@3). - CONFIG makes the pipeline reusable — change two lines to switch topic and audience.

Flow diagram:

```
config  (algorithm, target_audience, quality_threshold)
python >algorithm, >target_audience, >status, >iteration, >quality, >explanat
ion
[WHILE status:running]
  python <iteration >iteration
  @1 <algorithm <target_audience >explanation      (generate)
  @2 <explanation <target_audience >quality        (score 1-10)
  python <iteration <quality >status               (log + check)
[END_WHILE]
@3 <explanation >final_polish                      (polish)
python <final_polish <quality <iteration           (print)
```

Code:

```
algorithm=binary search
target_audience=first-year computer science student
quality_threshold=7

python >algorithm print("binary search")

python >target_audience print("first-year computer science student")

python >status print("running")

python >iteration print("0")

python >quality print("0")

python >explanation print("")

python <iteration >iteration import os val =
int(open(os.environ['MSH_VAR_iteration']).read().strip()) print(val + 1)

python <iteration <quality >status import os itr =
open(os.environ['MSH_VAR_iteration']).read().strip() q =
open(os.environ['MSH_VAR_quality']).read().strip() print(f"[Iter {itr}]
Quality: {q}") try:     print("done" if int(q) >= 7 else "running") except
ValueError:     print("running")

python <final_polish <quality <iteration import os q =
open(os.environ['MSH_VAR_quality']).read().strip() itr =
open(os.environ['MSH_VAR_iteration']).read().strip() print(f"=== Final
(score={q}, iters={itr}) ===")
print(open(os.environ['MSH_VAR_final_polish']).read().strip())
```

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

What it does: Python creates a list of 5 Python module names — some valid, some broken. FOREACH iterates. Per item TRY attempts `importlib.import_module`. If success, Python stores module info and LLM describes its use. If import fails, CATCH prints `[ERR]` as a hardcoded literal. Loop continues regardless of errors.

Key patterns: - TRY/CATCH inside FOREACH = per-item error isolation. - CATCH block does not use `<module_error` — prints the literal string directly. - Do not create dependencies on variables written inside TRY — if TRY fails they are not written.

Flow diagram:

```
python >modules = "os\ncollections\nfakemodule_xyz\njson\nnonexistent_pkg"
[FOREACH module in modules]
  [TRY]
    python <module >module_info       (importlib.import_module — may fail)
    @1 <module_info >insight          (LLM: describe the module)
    python <insight                   (print [OK])
```

```
  [CATCH >module_error]
    python                              (print [ERR] literal string)
  [END_TRY]
[END_FOREACH]
python  (completion message)
```

Code:

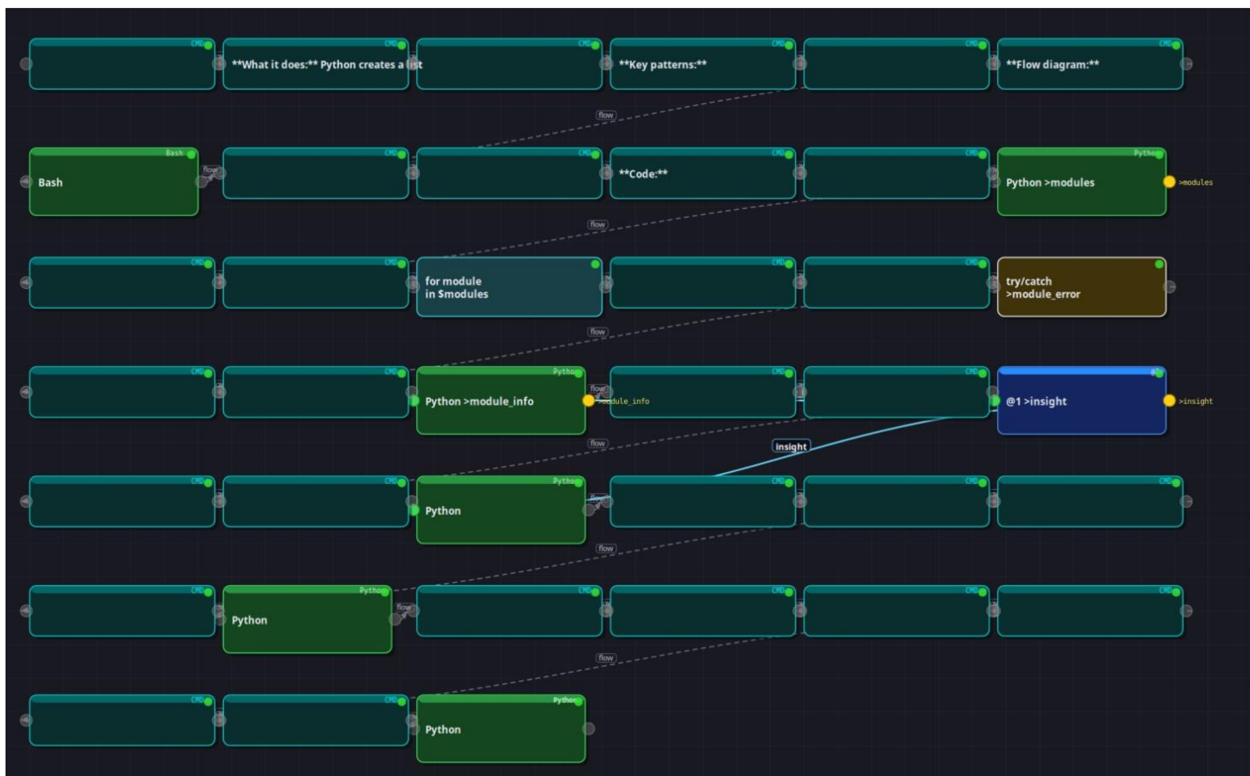
```
python >modules
print("os\ncollections\nfakemodule_xyz\njson\nnonexistent_pkg")
```

```
python <module >module_info import os, importlib module_name =
open(os.environ['MSH_VAR_module']).read().strip() mod =
importlib.import_module(module_name) doc_line = (mod.__doc__ or 'No
docstring').strip().split('\n')[0][:80] attrs = len([a for a in dir(mod) if
not a.startswith('_')]) print(f"Module: {module_name} | Public attrs: {attrs}
| Doc: {doc_line}")
```

```
python <insight import os print(f"[OK]
{open(os.environ['MSH_VAR_insight']).read().strip()}")
```

```
print("[ERR] Import failed: try_block_failed")
```

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



Complete Summary Table

#	Pattern	New Nodes	Models	Key Feature	Python Role
1	Linear Pipeline	—	—	Sequential data flow	All stages: generate → transform → report
2	LLM in Middle	—	@1	LLM as transformer	Generate data + parse LLM output
3	Fan-Out	—	@1	One source, many consumers	Lexical + structural + semantic analysis
4	Code Gen + Exec	—	@1	LLM generates executable code	Define task + execute via subprocess
5	Two-LLM Review	—	@1, @2	Generate → Review → Improve	Execute + validate final code
6	Parallel 3 Models	—	@1–@3	Same query to all models	Compare + display all responses
7	Eval-Optimizer Loop	LOOP	@1, @2	Loop until accepted	Execute + validate accepted code
8	Multi-Stage Pipeline	—	@1, @2	Full combination	Dataset gen + stats + final report
9	Routing	—	@1	Conditional branching	Classification + branch execution
10	Full Pipeline	—	@1–@3	All patterns combined	Primes + stats + format output
11	Chained LLM Nodes	—	@1, @2	Chained AI calls	Variable injection + metadata parsing
12	Async + Await + Synthesis	AWAIT	@1–@3	Parallel async + barrier	Code review + formatted report
13	WHILE Counter	WHILE	@1	Flag-based exit + LLM in loop	Fibonacci + multi-var file

#	Pattern	New Nodes	Models	Key Feature	Python Role
					writes
14	FOREACH List	FOREACH	@1	Line-by-line list processing	Library descriptions
15	TRY/CATCH	TRY/CATCH	—	Error isolation	JSON parsing with regex fallback
16	SPLIT + MERGE	SPLIT, MERGE	@1, @2	Async parallel + synthesis	Climate data analysis
17	CONFIG Pipeline	CONFIG	@1, @2	Parameterized template	Domain explanation report
18	FOREACH + Async	FOREACH, AWAIT	@1, @2	Per-item parallel LLM	Design pattern batch processing
19	WHILE Quality Gate	WHILE	@1, @2	Numeric threshold exit	Code generation until score >= 8
20	Map-Reduce	SPLIT, MERGE, AWAIT	@1, @2	Async map + reduce	Python docs concept extraction
21	TRY/CATCH + LOOP	TRY/CATCH, LOOP	@1	Retry with self-correction	Execute + catch + regenerate
22	Multi-Var Output	—	@1, @2	Structured field extraction	Regex parse → MSH_VAR_* writes
23	CONFIG+WHILE+3M	CONFIG, WHILE	@1–@3	Fully adaptive pipeline	Algorithm explanation + scoring
24	FOREACH+TRY/CATCH	FOREACH, TRY/CATCH	@1	Fault-tolerant batch	Module import with error isolation

---

Python Quick Reference

Reading a variable

```python
import os
value = open(os.environ['MSH_VAR_myvar']).read().strip()
```

Writing multiple output variables

```python
import os
with open(os.environ['MSH_VAR_var1'], 'w') as f: f.write(value1)
with open(os.environ['MSH_VAR_var2'], 'w') as f: f.write(value2)
```

Executing generated code safely

```
import os, subprocess, sys
path = os.environ['MSH_VAR_code']
result = subprocess.run([sys.executable, path], capture_output=True, text=Tru
e)
print(result.stdout)
if result.stderr: print("STDERR:", result.stderr)
```

WHILE loop control pattern

```
# Body must write to >status every iteration
import os
with open(os.environ['MSH_VAR_status'], 'w') as f:
    f.write("done" if condition else "running")
```

FOREACH list creation

```
# Use print with \n — NOT space-separated
print("item1\nitem2\nitem3")
```

TRY/CATCH: CATCH block

```
# NEVER read <errvar inside CATCH
# Always print the literal string
print("[ERR] Something failed: try_block_failed")
```

Multi-variable output from Python block

```
import os, re
# Write each field directly to the MSH_VAR_* path
with open(os.environ['MSH_VAR_summary'], 'w') as f: f.write(summary)
with open(os.environ['MSH_VAR_tags'], 'w') as f: f.write(tags)
# NOTE: stdout is NOT captured when block has multiple >outvar
```

Appendix I: Code examples for each pattern

/home/igor > Sent to mshell (1517 bytes)

Received from GUI editor:

Pattern 1: Linear Data Pipeline (Python)

Sequential data flow: generate \u2192 validate \u2192 transform \u2192 format \u2192 report

```python >raw_numbers

import random

random.seed(42)
```

````
numbers = [random.randint(1, 100) for _ in range(12)]
print(','.join(map(str, numbers)))
```

```python <raw_numbers >stats
import os
data = open(os.environ['MSH_VAR_raw_numbers']).read().strip()
nums = list(map(int, data.split(',')))
mean = sum(nums) / len(nums)
variance = sum((x - mean) ** 2 for x in nums) / len(nums)
std = variance ** 0.5
print(f"count={len(nums)} sum={sum(nums)} mean={mean:.2f} std={std:.2f} min={min(nums)} max={max(nums)}")
```

```python <stats >filtered
import os
stats = open(os.environ['MSH_VAR_stats']).read().strip()
parts = dict(p.split('=') for p in stats.split())
mean = float(parts['mean'])
std = float(parts['std'])
print(f"Threshold analysis: values beyond mean±1\u03c3 = [{mean - std:.1f}, {mean + std:.1f}]")
print(f"Distribution: {stats}")
```

```python <filtered >formatted
import os
text = open(os.environ['MSH_VAR_filtered']).read().strip()
lines = text.split('\n')
formatted = '\n'.join(f"  -> {line}" for line in lines)
print("=== Pipeline Stage 4: Formatted Report ===")
````

print(formatted)

```

```python &lt;formatted &gt;report

import os

content = open(os.environ['MSH_VAR_formatted']).read().strip()

report = "=== DATA PIPELINE FINAL REPORT ===\n" + content + "\n[Pipeline completed successfully]"

print(report)

```

```python <report

import os

print(open(os.environ['MSH_VAR_report']).read())

```

82,15,4,95,36,32,29,18,95,14,87,95

count=12 sum=602 mean=50.17 std=35.46 min=4 max=95

Threshold analysis: values beyond mean±1\u03c3 = [14.7, 85.6]

Distribution: count=12 sum=602 mean=50.17 std=35.46 min=4 max=95

=== Pipeline Stage 4: Formatted Report ===

  -> Threshold analysis: values beyond mean±1\u03c3 = [14.7, 85.6]

  -> Distribution: count=12 sum=602 mean=50.17 std=35.46 min=4 max=95

=== DATA PIPELINE FINAL REPORT ===

=== Pipeline Stage 4: Formatted Report ===

  -> Threshold analysis: values beyond mean±1\u03c3 = [14.7, 85.6]

  -> Distribution: count=12 sum=602 mean=50.17 std=35.46 min=4 max=95

[Pipeline completed successfully]

/home/igor > === DATA PIPELINE FINAL REPORT ===

=== Pipeline Stage 4: Formatted Report ===
```

-> Threshold analysis: values beyond mean±1\u03c3 = [14.7, 85.6]

-> Distribution: count=12 sum=602 mean=50.17 std=35.46 min=4 max=95

[Pipeline completed successfully]

/home/igor >

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

/home/igor > Sent to mshell (1240 bytes)

Received from GUI editor:

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

# Pattern 2: LLM in the Middle (Python)

# Python generates structured data \u2192 LLM analyzes \u2192 Python processes result

```python >market_data
data = {
    "ticker": "TECHX",
    "week": ["Mon", "Tue", "Wed", "Thu", "Fri"],
    "prices": [142.5, 138.2, 145.8, 151.3, 149.7],
    "volume": [2.1, 3.4, 1.8, 2.9, 2.2]
}
lines = [f"{d}: price=${p:.1f}, vol={v:.1f}M" for d, p, v in zip(data['week'], data['prices'], data['volume'])]
print(f"Ticker: {data['ticker']}")
print('\n'.join(lines))
```

<!--@1 <market_data >analysis

The input contains daily stock data for a ticker.

Write a concise analyst report: identify the trend, best and worst day, and give a one-sentence outlook. Two paragraphs max.

-->

```python <analysis
import os
```

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

words = text.split()

sentences = sum(1 for c in text if c in '.!?')

positive = sum(1 for w in words if w.lower() in
['growth','strong','up','gain','positive','rising','increased'])

negative = sum(1 for w in words if w.lower() in
['drop','weak','down','loss','negative','falling','decreased'])

print(f"[Meta] Words: {len(words)} | Sentences: {sentences} | +signals: {positive} | -signals:
{negative}")

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

print(text)
```

Ticker: TECHX

Mon: price=$142.5, vol=2.1M

Tue: price=$138.2, vol=3.4M

Wed: price=$145.8, vol=1.8M

Thu: price=$151.3, vol=2.9M

Fri: price=$149.7, vol=2.2M

TECHX Weekly Analysis:

TECHX exhibited strong bullish momentum this week, gaining 5.1% from Monday's $142.50 to Friday's close of $149.70. Thursday was the best performing day with a 3.8% surge to $151.30 on elevated volume of 2.9M shares, while Tuesday marked the week's low at $138.20 (-3.0%) but with the highest volume of 3.4M, suggesting institutional accumulation on the dip.

Outlook: The stock demonstrates healthy upward momentum with strong support above $145 and appears poised for further gains if it can break through the $151-152 resistance level established on Thursday's high.

/home/igor > [Meta] Words: 91 | Sentences: 12 | +signals: 2 | -signals: 0

--- LLM Analysis ---

TECHX Weekly Analysis:

TECHX exhibited strong bullish momentum this week, gaining 5.1% from Monday's $142.50 to Friday's close of $149.70. Thursday was the best performing day with a 3.8% surge to $151.30 on elevated volume of 2.9M shares, while Tuesday marked the week's low at $138.20 (-3.0%) but with the highest volume of 3.4M, suggesting institutional accumulation on the dip.

Outlook: The stock demonstrates healthy upward momentum with strong support above $145 and appears poised for further gains if it can break through the $151-152 resistance level established on Thursday's high.

/home/igor >

/home/igor > Sent to mshell (1563 bytes)

Received from GUI editor:

Pattern 3: Fan-Out One Variable, Many Consumers (Python)

Single data source consumed by multiple independent Python blocks + LLM

```python >corpus

text = (

    "Artificial intelligence is transforming every industry. "

    "From healthcare to finance, machine learning models detect patterns humans cannot see. "

    "Natural language processing enables computers to understand context and nuance. "

    "Neural networks learn hierarchical features from raw data automatically. "

    "The future belongs to those who can harness these powerful tools responsibly."

)

print(text.strip())

```

```python <corpus

import os, re

text = open(os.environ['MSH_VAR_corpus']).read().strip()

words = text.split()

unique = set(w.lower().strip('.,!?') for w in words)
```

```
avg_word_len = sum(len(w) for w in words) / len(words)

print(f"[Lexical] Total words: {len(words)} | Unique: {len(unique)} | Avg word length: {avg_word_len:.1f}")
```

```python <corpus
import os

text = open(os.environ['MSH_VAR_corpus']).read().strip()

sentences = [s.strip() for s in text.split('.') if s.strip()]

lengths = [len(s.split()) for s in sentences]

print(f"[Structure] Sentences: {len(sentences)} | Avg length: {sum(lengths)/len(lengths):.1f} words | Longest: {max(lengths)} words")
```

<!--@1 <corpus >semantics

The input is a short text passage. Identify the main domain, 3 key concepts, and the overall tone.

Reply in 3 lines: Domain: / Concepts: / Tone:

-->

```python <semantics
import os

print("[Semantic Analysis]")

print(open(os.environ['MSH_VAR_semantics']).read().strip())
```

Artificial intelligence is transforming every industry. From healthcare to finance, machine learning models detect patterns humans cannot see. Natural language processing enables computers to understand context and nuance. Neural networks learn hierarchical features from raw data automatically. The future belongs to those who can harness these powerful tools responsibly.

[Lexical] Total words: 49 | Unique: 46 | Avg word length: 6.6

[Structure] Sentences: 5 | Avg length: 9.8 words | Longest: 12 words

Domain: Artificial Intelligence and Technology

Concepts: Machine learning pattern detection, Natural language processing, Neural networks and automated feature learning

Tone: Optimistic and forward-looking with emphasis on responsibility

/home/igor > [Semantic Analysis]

Domain: Artificial Intelligence and Technology

Concepts: Machine learning pattern detection, Natural language processing, Neural networks and automated feature learning

Tone: Optimistic and forward-looking with emphasis on responsibility

/home/igor >

/home/igor > Sent to mshell (966 bytes)

Received from GUI editor:

Pattern 4: LLM Code Generation - Execute via Variable (Python)

LLM generates executable Python code \u2192 Python inspects and runs it

```python >task
task = (
    "Write a Python function called analyze_text(text) that counts word frequency "
    "and returns the top 5 most common words with their counts. "
    "Call it with: 'to be or not to be that is the question'. "
    "Print results as: word: count"
)
print(task.strip())
```

<!--@1 <task >code

The input is a Python coding task.

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

The code must run standalone and print its output.

-->

```python &lt;code
import os, subprocess, sys

path = os.environ['MSH_VAR_code']

print("=== Generated Code ===")

print(open(path).read())

print("=== Execution Output ===")

result = subprocess.run([sys.executable, path], capture_output=True, text=True)

print(result.stdout)

if result.stderr:

    print("STDERR:", result.stderr)
```

Write a Python function called analyze_text(text) that counts word frequency and returns the top 5 most common words with their counts. Call it with: 'to be or not to be that is the question'. Print results as: word: count

```
def analyze_text(text):

    words = text.lower().split()

    word_count = {}

    for word in words:

        word_count[word] = word_count.get(word, 0) + 1

    sorted_words = sorted(word_count.items(), key=lambda x: x[1], reverse=True)

    return sorted_words[:5]

result = analyze_text('to be or not to be that is the question')

for word, count in result:

    print(f"{word}: {count}")

/home/igor > === Generated Code ===

def analyze_text(text):

    words = text.lower().split()
```

```
    word_count = {}
    for word in words:
        word_count[word] = word_count.get(word, 0) + 1
    sorted_words = sorted(word_count.items(), key=lambda x: x[1], reverse=True)
    return sorted_words[:5]
result = analyze_text('to be or not to be that is the question')
for word, count in result:
    print(f"{word}: {count}")
```

=== Execution Output ===

to: 2

be: 2

or: 1

not: 1

that: 1

/home/igor >

/home/igor > Sent to mshell (1475 bytes)

Received from GUI editor:

Pattern 5: Two-LLM Review Chain (Python)

Model 1 generates code \u2192 Model 2 reviews \u2192 Model 1 improves \u2192 Python executes

```python >task
print("Write a Python function `moving_average(data, window)` that computes the moving average of a list. Handle edge cases: window larger than data, empty list, window=1. Include a demo with data=[1,3,5,7,9,11,13] and window=3.")
```

<!--@1 <task >code

Write only Python code. No markdown fences. No explanation.

Include the function and a demo that prints results.

-->

```python <code
import os
print("=== Model 1: Initial Code ===")
print(open(os.environ['MSH_VAR_code']).read())
```

<!--@2 <code >review

Review this Python code. Check: correctness, edge case handling, Pythonic style, efficiency.

Reply in 3 sentences: what is correct, what can be improved, and one specific fix to apply.

-->

```python <review
import os
print("=== Model 2: Code Review ===")
print(open(os.environ['MSH_VAR_review']).read())
```

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

The first input is Python code. The second input is a code review.

Apply the feedback and return ONLY the improved Python code, no fences.

-->

```python <improved
import os, subprocess, sys
path = os.environ['MSH_VAR_improved']
print("=== Final Improved Code ===")
print(open(path).read())
print("=== Execution ===")
result = subprocess.run([sys.executable, path], capture_output=True, text=True)
print(result.stdout)
```

```
if result.stderr:
    print("STDERR:", result.stderr)
```

Write a Python function `moving_average(data, window)` that computes the moving average of a list. Handle edge cases: window larger than data, empty list, window=1. Include a demo with data=[1,3,5,7,9,11,13] and window=3.

```
def moving_average(data, window):
    if not data or window <= 0:
        return []
    if window > len(data):
        return [sum(data) / len(data)]
    if window == 1:
        return data[:]
    result = []
    for i in range(len(data) - window + 1):
        avg = sum(data[i:i + window]) / window
        result.append(avg)
    return result
data = [1, 3, 5, 7, 9, 11, 13]
window = 3
result = moving_average(data, window)
print(f"Data: {data}")
print(f"Window: {window}")
print(f"Moving averages: {result}")


print(f"Empty list test: {moving_average([], 3)}")
print(f"Window larger than data: {moving_average([1, 2], 5)}")
print(f"Window=1 test: {moving_average([1, 2, 3], 1)}")
```

=== Model 1: Initial Code ===

```
def moving_average(data, window):
    if not data or window <= 0:
        return []
    if window > len(data):
        return [sum(data) / len(data)]
    if window == 1:
        return data[:]
    result = []
    for i in range(len(data) - window + 1):
        avg = sum(data[i:i + window]) / window
        result.append(avg)
    return result
data = [1, 3, 5, 7, 9, 11, 13]
window = 3
result = moving_average(data, window)
print(f"Data: {data}")
print(f"Window: {window}")
print(f"Moving averages: {result}")
print(f"Empty list test: {moving_average([], 3)}")
print(f"Window larger than data: {moving_average([1, 2], 5)}")
print(f"Window=1 test: {moving_average([1, 2, 3], 1)}")
```

It's logically correct, handles the specified edge cases well, and produces the expected moving averages. Style is mostly Pythonic, though multiple early returns and recomputing `sum` on each sliding window make it less efficient than it could be. To improve efficiency, precompute the first window sum and update it incrementally, e.g. maintain a running sum instead of calling `sum(data[i:i + window])` inside the loop.

=== Model 2: Code Review ===

It's logically correct, handles the specified edge cases well, and produces the expected moving averages. Style is mostly Pythonic, though multiple early returns and recomputing

`sum` on each sliding window make it less efficient than it could be. To improve efficiency, precompute the first window sum and update it incrementally, e.g. maintain a running sum instead of calling `sum(data[i:i + window])` inside the loop.

```
def moving_average(data, window):
    if not data or window <= 0:
        return []
    if window > len(data):
        return [sum(data) / len(data)]
    if window == 1:
        return data[:]
    result = []
    window_sum = sum(data[:window])
    result.append(window_sum / window)
    for i in range(1, len(data) - window + 1):
        window_sum = window_sum - data[i - 1] + data[i + window - 1]
        result.append(window_sum / window)
    return result
data = [1, 3, 5, 7, 9, 11, 13]
window = 3
result = moving_average(data, window)
print(f"Data: {data}")
print(f"Window: {window}")
print(f"Moving averages: {result}")
print(f"Empty list test: {moving_average([], 3)}")
print(f"Window larger than data: {moving_average([1, 2], 5)}")
print(f"Window=1 test: {moving_average([1, 2, 3], 1)}")
/home/igor > === Final Improved Code ===
def moving_average(data, window):
    if not data or window <= 0:
```

```
        return []
    if window > len(data):
        return [sum(data) / len(data)]
    if window == 1:
        return data[:]
    result = []
    window_sum = sum(data[:window])
    result.append(window_sum / window)
    for i in range(1, len(data) - window + 1):
        window_sum = window_sum - data[i - 1] + data[i + window - 1]
        result.append(window_sum / window)
    return result
data = [1, 3, 5, 7, 9, 11, 13]
window = 3
result = moving_average(data, window)
print(f"Data: {data}")
print(f"Window: {window}")
print(f"Moving averages: {result}")
print(f"Empty list test: {moving_average([], 3)}")
print(f"Window larger than data: {moving_average([1, 2], 5)}")
print(f"Window=1 test: {moving_average([1, 2, 3], 1)}")
```

=== Execution ===

```
Data: [1, 3, 5, 7, 9, 11, 13]
Window: 3
Moving averages: [3.0, 5.0, 7.0, 9.0, 11.0]
Empty list test: []
Window larger than data: [1.5]
Window=1 test: [1, 2, 3]
```

/home/igor >

/home/igor > Sent to mshell (1046 bytes)

Received from GUI editor:

Pattern 6: Parallel 3-Model Query (Python)

Same question sent to all 3 LLM models \u2192 Python collects and compares

```python >question

print("Given a dataset of 1 million rows, how would you efficiently find all duplicate records in Python? Describe your data structure choice, time complexity, and Python-specific optimizations. Answer in 3-5 sentences.")

```

<!--@1 <question >ans1

Answer concisely and technically. 3-5 sentences.

-->

<!--@2 <question >ans2

Answer with practical Python library recommendations. 3-5 sentences.

-->

<!--@3 <question >ans3

Answer focusing on memory efficiency and scalability. 3-5 sentences.

-->

```python <ans1 <ans2 <ans3

import os

answers = {

   'Model 1 (Technical)': open(os.environ['MSH_VAR_ans1']).read().strip(),

   'Model 2 (Libraries)': open(os.environ['MSH_VAR_ans2']).read().strip(),

   'Model 3 (Scalability)': open(os.environ['MSH_VAR_ans3']).read().strip(),

}

for label, text in answers.items():
```

```
    print(f"=== {label} ({len(text.split())} words) ===")
    print(text)
    print()
```

Given a dataset of 1 million rows, how would you efficiently find all duplicate records in Python? Describe your data structure choice, time complexity, and Python-specific optimizations. Answer in 3-5 sentences.

Use a Python dictionary or `collections.Counter` to hash each record, where keys are tuples of the record values and values are lists of row indices or occurrence counts, achieving O(n) average time complexity. For memory efficiency with large datasets, consider using `pandas.DataFrame.duplicated()` or `pandas.groupby()` which leverage optimized C implementations underneath. Python-specific optimizations include using `frozenset` for unordered duplicate detection, `itertools.groupby()` after sorting for ordered data, or `collections.defaultdict(list)` to avoid key existence checks. For extremely large datasets, implement a two-pass approach: first pass builds a set of seen records, second pass identifies duplicates, reducing memory overhead from O(n) to O(d) where d is the number of unique records.

Use a hash-based structure: represent each row (or a key subset of columns) as a tuple and track frequencies in a `dict` or `collections.Counter`, giving average O(n) time and O(n) space. For tabular data (CSV, Parquet, SQL exports), `pandas` is usually best: `df[df.duplicated(keep=False)]` returns all duplicate rows using fast, C-optimized hashing. When loading with pandas, specify `dtype`, `usecols`, and possibly `category` for repeated strings to reduce memory and speed hashing. If 1M rows is near your RAM limit, use `pandas.read_csv(..., chunksize=...)` plus a `Counter` across chunks, or scale further with `dask.dataframe` which mirrors the pandas API but works out-of-core.

Use a `set` to track seen records and another `set` for duplicates: iterate once, adding unseen records to `seen`, and duplicates to `duplicates` when encountered again. This achieves O(n) average time complexity with O(n) memory, optimal for 1M rows. Prioritize hashable record representations (e.g., tuples) for fast set operations in Python. For scalability, process records iteratively from disk (e.g., via `csv.reader`) to avoid loading all data into memory at once. (58 words)

/home/igor > === Model 1 (Technical) (107 words) ===

Use a Python dictionary or `collections.Counter` to hash each record, where keys are tuples of the record values and values are lists of row indices or occurrence counts, achieving O(n) average time complexity. For memory efficiency with large datasets, consider using `pandas.DataFrame.duplicated()` or `pandas.groupby()` which leverage optimized C implementations underneath. Python-specific optimizations include using `frozenset` for unordered duplicate detection, `itertools.groupby()` after sorting for ordered data, or

`collections.defaultdict(list)` to avoid key existence checks. For extremely large datasets, implement a two-pass approach: first pass builds a set of seen records, second pass identifies duplicates, reducing memory overhead from O(n) to O(d) where d is the number of unique records.

=== Model 2 (Libraries) (99 words) ===

Use a hash-based structure: represent each row (or a key subset of columns) as a tuple and track frequencies in a `dict` or `collections.Counter`, giving average O(n) time and O(n) space. For tabular data (CSV, Parquet, SQL exports), `pandas` is usually best: `df[df.duplicated(keep=False)]` returns all duplicate rows using fast, C-optimized hashing. When loading with pandas, specify `dtype`, `usecols`, and possibly `category` for repeated strings to reduce memory and speed hashing. If 1M rows is near your RAM limit, use `pandas.read_csv(..., chunksize=...)` plus a `Counter` across chunks, or scale further with `dask.dataframe` which mirrors the pandas API but works out-of-core.

=== Model 3 (Scalability) (72 words) ===

Use a `set` to track seen records and another `set` for duplicates: iterate once, adding unseen records to `seen`, and duplicates to `duplicates` when encountered again. This achieves O(n) average time complexity with O(n) memory, optimal for 1M rows. Prioritize hashable record representations (e.g., tuples) for fast set operations in Python. For scalability, process records iteratively from disk (e.g., via `csv.reader`) to avoid loading all data into memory at once. (58 words)

/home/igor >

/home/igor > Sent to mshell (1465 bytes)

Received from GUI editor:

Pattern 7: Evaluator-Optimizer Loop (Python)

Generator \u2192 Evaluator \u2192 loop until ACCEPTED or max iterations

```python >task

print("Write a Python function `validate_email_list(emails)` that takes a list of strings and returns a dict with keys 'valid' and 'invalid', each containing the respective emails. Use regex. Must handle None and empty strings gracefully.")

```

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

<!--@1 <task >code

Write only Python code. No markdown fences. No explanation.

Include the function and a demo call with valid emails, None, empty strings, and malformed inputs.

-->

```python <code
import os

print("=== Generated Code ===")

print(open(os.environ['MSH_VAR_code']).read())
```

<!--@2 <code >verdict

Review this Python code strictly.

Check: regex correctness, None handling, empty string handling, return structure.

If fully correct: reply with exactly ACCEPTED on the first line.

If not: reply with REJECTED on the first line, then one sentence explaining what to fix.

-->

```python <verdict
import os

print("=== Verdict ===", open(os.environ['MSH_VAR_verdict']).read().strip())
```

<!--@end_loop-->

```python <code
import os, subprocess, sys

path = os.environ['MSH_VAR_code']

print("=== Final Accepted Code ===")

print(open(path).read())

print("=== Execution ===")

result = subprocess.run([sys.executable, path], capture_output=True, text=True)

print(result.stdout)

if result.stderr:
```

print("STDERR:", result.stderr)

```

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

Write a Python function `validate_email_list(emails)` that takes a list of strings and returns a dict with keys 'valid' and 'invalid', each containing the respective emails. Use regex. Must handle None and empty strings gracefully.

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

```
import re

def validate_email_list(emails):
    if not emails:
        return {'valid': [], 'invalid': []}
    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    valid = []
    invalid = []
    for email in emails:
        if email is None or email == '':
            invalid.append(email)
        elif re.match(email_pattern, email):
            valid.append(email)
        else:
            invalid.append(email)
    return {'valid': valid, 'invalid': invalid}
test_emails = [
    'user@example.com',
    'valid.email+tag@domain.co.uk',
    'test@test.org',
    None,
    '',
    'invalid.email',
```
```

```
    '@domain.com',
    'user@',
    'spaces in@email.com',
    'user@domain',
    'another.valid@email-domain.net'
]
result = validate_email_list(test_emails)
print(f"Valid emails: {result['valid']}")
print(f"Invalid emails: {result['invalid']}")
```

=== Generated Code ===

```
import re
def validate_email_list(emails):
    if not emails:
        return {'valid': [], 'invalid': []}
    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    valid = []
    invalid = []
    for email in emails:
        if email is None or email == '':
            invalid.append(email)
        elif re.match(email_pattern, email):
            valid.append(email)
        else:
            invalid.append(email)
    return {'valid': valid, 'invalid': invalid}
test_emails = [
    'user@example.com',
    'valid.email+tag@domain.co.uk',
```

```
    'test@test.org',
    None,
    '',
    'invalid.email',
    '@domain.com',
    'user@',
    'spaces in@email.com',
    'user@domain',
    'another.valid@email-domain.net'
]
result = validate_email_list(test_emails)
print(f"Valid emails: {result['valid']}")
print(f"Invalid emails: {result['invalid']}")
```

REJECTED

The regex is too strict to be considered fully correct for email validation because it disallows valid addresses with top-level domains of length 1 (e.g., local/test cases) and other RFC-valid edge cases, so you should either relax the pattern or use a dedicated email validation library.

[loop] until check: last_line='The regex is too strict to be considered fully correct for email validation because it disallows valid addresses with top-level domains of length 1 (e.g., local/test cases) and other RFC-valid edge cases, so you should either relax the pattern or use a de' expected='ACCEPTED'

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

=== Verdict === REJECTED

The regex is too strict to be considered fully correct for email validation because it disallows valid addresses with top-level domains of length 1 (e.g., local/test cases) and other RFC-valid edge cases, so you should either relax the pattern or use a dedicated email validation library.

```
import re
def validate_email_list(emails):
    if not emails:
```

```
        return {'valid': [], 'invalid': []}
    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    valid = []
    invalid = []
    for email in emails:
        if email is None or email == '':
            invalid.append(email)
        elif re.match(email_pattern, email):
            valid.append(email)
        else:
            invalid.append(email)
    return {'valid': valid, 'invalid': invalid}
test_emails = [
    'user@example.com',
    'valid.email+tag@domain.co.uk',
    'test@test.org',
    None,
    '',
    'invalid.email',
    '@domain.com',
    'user@',
    'spaces in@email.com',
    'user@domain',
    'another.valid@email-domain.net'
]
result = validate_email_list(test_emails)
print(f"Valid emails: {result['valid']}")
print(f"Invalid emails: {result['invalid']}")
```

=== Generated Code ===

```
import re
def validate_email_list(emails):
    if not emails:
        return {'valid': [], 'invalid': []}
    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    valid = []
    invalid = []
    for email in emails:
        if email is None or email == '':
            invalid.append(email)
        elif re.match(email_pattern, email):
            valid.append(email)
        else:
            invalid.append(email)
    return {'valid': valid, 'invalid': invalid}
test_emails = [
    'user@example.com',
    'valid.email+tag@domain.co.uk',
    'test@test.org',
    None,
    '',
    'invalid.email',
    '@domain.com',
    'user@',
    'spaces in@email.com',
    'user@domain',
    'another.valid@email-domain.net'
```

```
]

result = validate_email_list(test_emails)

print(f"Valid emails: {result['valid']}")

print(f"Invalid emails: {result['invalid']}")
```

REJECTED

The regex is oversimplified and not fully correct for email validation (it rejects some RFC-valid emails and can accept invalid ones, e.g., with consecutive dots or leading/trailing dots in the local/domain parts), so you should either tighten the pattern or use a dedicated validation library like `email_validator`.

[loop] until check: last_line='The regex is oversimplified and not fully correct for email validation (it rejects some RFC-valid emails and can accept invalid ones, e.g., with consecutive dots or leading/trailing dots in the local/domain parts), so you should either tighten the pattern' expected='ACCEPTED'

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

=== Verdict === REJECTED

The regex is oversimplified and not fully correct for email validation (it rejects some RFC-valid emails and can accept invalid ones, e.g., with consecutive dots or leading/trailing dots in the local/domain parts), so you should either tighten the pattern or use a dedicated validation library like `email_validator`.

```
import re

def validate_email_list(emails):

    if not emails:

        return {'valid': [], 'invalid': []}

    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    valid = []

    invalid = []

    for email in emails:

        if email is None or email == '':

            invalid.append(email)

        elif re.match(email_pattern, email):

            valid.append(email)
```

```
        else:
            invalid.append(email)
    return {'valid': valid, 'invalid': invalid}
test_emails = [
    'user@example.com',
    'valid.email+tag@domain.co.uk',
    'test@test.org',
    None,
    '',
    'invalid.email',
    '@domain.com',
    'user@',
    'spaces in@email.com',
    'user@domain',
    'another.valid@email-domain.net'
]
result = validate_email_list(test_emails)
print(f"Valid emails: {result['valid']}")
print(f"Invalid emails: {result['invalid']}")
=== Generated Code ===
import re
def validate_email_list(emails):
    if not emails:
        return {'valid': [], 'invalid': []}
    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    valid = []
    invalid = []
    for email in emails:
```

```
        if email is None or email == '':
            invalid.append(email)
        elif re.match(email_pattern, email):
            valid.append(email)
        else:
            invalid.append(email)
    return {'valid': valid, 'invalid': invalid}
test_emails = [
    'user@example.com',
    'valid.email+tag@domain.co.uk',
    'test@test.org',
    None,
    '',
    'invalid.email',
    '@domain.com',
    'user@',
    'spaces in@email.com',
    'user@domain',
    'another.valid@email-domain.net'
]
result = validate_email_list(test_emails)
print(f"Valid emails: {result['valid']}")
print(f"Invalid emails: {result['invalid']}")
```

REJECTED

The regex is an oversimplification of valid email formats and can both reject some RFC-compliant addresses and accept invalid ones (e.g., with consecutive or leading/trailing dots), so you should either tighten the pattern or use Python\u2019s `email.utils.parseaddr` or a dedicated library like `email_validator` for robust validation.

[loop] until check: last_line='The regex is an oversimplification of valid email formats and can both reject some RFC-compliant addresses and accept invalid ones (e.g., with

consecutive or leading/trailing dots), so you should either tighten the pattern or use Python\u2019s `email.utils.p' expected='ACCEPTED'

[loop] Exiting loop after 3 iteration(s). reason: max iterations reached

=== Verdict === REJECTED

The regex is an oversimplification of valid email formats and can both reject some RFC-compliant addresses and accept invalid ones (e.g., with consecutive or leading/trailing dots), so you should either tighten the pattern or use Python\u2019s `email.utils.parseaddr` or a dedicated library like `email_validator` for robust validation.

/home/igor > === Final Accepted Code ===

```
import re

def validate_email_list(emails):

    if not emails:

        return {'valid': [], 'invalid': []}

    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    valid = []

    invalid = []

    for email in emails:

        if email is None or email == '':

            invalid.append(email)

        elif re.match(email_pattern, email):

            valid.append(email)

        else:

            invalid.append(email)

    return {'valid': valid, 'invalid': invalid}

test_emails = [

    'user@example.com',

    'valid.email+tag@domain.co.uk',

    'test@test.org',

    None,

    '',
```

'invalid.email',

'@domain.com',

'user@',

'spaces in@email.com',

'user@domain',

'another.valid@email-domain.net'

]

result = validate_email_list(test_emails)

print(f"Valid emails: {result['valid']}")

print(f"Invalid emails: {result['invalid']}")

=== Execution ===

Valid emails: ['user@example.com', 'valid.email+tag@domain.co.uk', 'test@test.org', 'another.valid@email-domain.net']

Invalid emails: [None, '', 'invalid.email', '@domain.com', 'user@', 'spaces in@email.com', 'user@domain']

/home/igor >

/home/igor > Sent to mshell (2245 bytes)

Received from GUI editor:

Pattern 8: Multi-Stage Python Analysis Pipeline (Python)

Python generates dataset \u2192 Python computes stats \u2192 LLM1 analyzes \u2192 LLM2 recommends \u2192 Python formats report

```python >raw_data

import random

random.seed(99)

depts = ['Engineering', 'Marketing', 'Sales', 'HR']

print("dept,salary,tenure,perf_score")

for i in range(20):
```

```
    dept = random.choice(depts)
    salary = random.randint(55000, 130000)
    tenure = random.randint(1, 15)
    score = round(random.uniform(2.5, 5.0), 1)
    print(f"{dept},{salary},{tenure},{score}")
```

```python <raw_data >stats
import os
from collections import defaultdict
lines = open(os.environ['MSH_VAR_raw_data']).read().strip().split('\n')
rows = [l.split(',') for l in lines[1:]]
dept_data = defaultdict(lambda: {'salaries': [], 'tenures': [], 'scores': []})
for dept, salary, tenure, score in rows:
    dept_data[dept]['salaries'].append(int(salary))
    dept_data[dept]['tenures'].append(int(tenure))
    dept_data[dept]['scores'].append(float(score))
for dept, d in sorted(dept_data.items()):
    avg_sal = sum(d['salaries']) / len(d['salaries'])
    avg_ten = sum(d['tenures']) / len(d['tenures'])
    avg_scr = sum(d['scores']) / len(d['scores'])
    print(f"{dept}: n={len(d['salaries'])} avg_salary=${avg_sal:.0f} avg_tenure={avg_ten:.1f}yr avg_score={avg_scr:.2f}")
```

<!--@1 <stats >analysis

The input contains department-level HR statistics.

Write one paragraph identifying key patterns, disparities, and notable findings across departments.

-->

<!--@2 <analysis >recommendations

The input is an HR analysis.

Provide 3 specific, actionable recommendations for management. Number each one.

-->

```python &lt;raw_data &lt;stats &lt;analysis &lt;recommendations
import os
raw = open(os.environ['MSH_VAR_raw_data']).read().strip().split('\n')
stats = open(os.environ['MSH_VAR_stats']).read().strip()
analysis = open(os.environ['MSH_VAR_analysis']).read().strip()
recs = open(os.environ['MSH_VAR_recommendations']).read().strip()
print("=" * 50)
print("      HR ANALYTICS MANAGEMENT REPORT")
print("=" * 50)
print(f"\n[Dataset] {len(raw)-1} employee records\n")
print("[Statistics]\n" + stats)
print("\n[Analysis]\n" + analysis)
print("\n[Recommendations]\n" + recs)
print("\n" + "=" * 50)
```

----

dept,salary,tenure,perf_score

HR,104906,4,4.0

Marketing,87562,3,4.4

Sales,105209,9,4.2

Engineering,119130,4,3.6

Marketing,104018,14,3.5

Marketing,74959,8,4.2

Sales,65874,12,3.8

Engineering,116477,6,3.5

Sales,128131,1,3.6

Sales,125033,2,3.7

Marketing,115647,1,3.4

Marketing,76077,12,4.5

Marketing,120954,11,2.9

Engineering,58054,3,3.7

HR,65637,15,3.6

Engineering,73731,5,4.0

HR,98214,3,3.8

Marketing,62261,11,3.0

Sales,126495,7,4.6

Marketing,101374,12,3.7

Auto-installing missing Python packages...

Engineering: n=4 avg_salary=$91848 avg_tenure=4.5yr avg_score=3.70

HR: n=3 avg_salary=$89586 avg_tenure=7.3yr avg_score=3.80

Marketing: n=8 avg_salary=$92856 avg_tenure=9.0yr avg_score=3.70

Sales: n=5 avg_salary=$110148 avg_tenure=6.2yr avg_score=3.98

The data reveals significant compensation and performance disparities across departments, with Sales commanding the highest average salary at $110,148 despite having mid-range tenure (6.2 years) and Marketing showing the strongest employee retention with 9.0 years average tenure but relatively lower compensation at $92,856. Sales also leads in performance scores (3.98), suggesting a strong correlation between compensation and results, while HR demonstrates exceptional loyalty with 7.3 years tenure and competitive performance (3.80) despite having the lowest salary ($89,586). Engineering, though having the shortest tenure (4.5 years) and smallest team size (n=4), maintains competitive compensation ($91,848), potentially indicating either high turnover or recent hiring in a competitive market.

1. Rebalance compensation for high-tenure, lower-paid functions (especially Marketing and HR) by conducting a market benchmark and introducing targeted salary adjustments or retention bonuses to reduce the risk of losing experienced, loyal staff.

2. Deep-dive into Engineering\u2019s short tenure to determine whether it\u2019s driven by turnover or recent growth, then respond with clearer career paths, mentorship, and possibly sign-on/retention incentives if the market is highly competitive.

3. Analyze the specific practices in Sales that link higher pay to higher performance (e.g., variable pay, incentives, goal-setting) and selectively adapt those mechanisms to other departments, ensuring transparent performance metrics so increased compensation clearly ties to measurable results.

/home/igor > ==

HR ANALYTICS MANAGEMENT REPORT

==

[Dataset] 20 employee records

[Statistics]

Engineering: n=4 avg_salary=$91848 avg_tenure=4.5yr avg_score=3.70

HR: n=3 avg_salary=$89586 avg_tenure=7.3yr avg_score=3.80

Marketing: n=8 avg_salary=$92856 avg_tenure=9.0yr avg_score=3.70

Sales: n=5 avg_salary=$110148 avg_tenure=6.2yr avg_score=3.98

[Analysis]

The data reveals significant compensation and performance disparities across departments, with Sales commanding the highest average salary at $110,148 despite having mid-range tenure (6.2 years) and Marketing showing the strongest employee retention with 9.0 years average tenure but relatively lower compensation at $92,856. Sales also leads in performance scores (3.98), suggesting a strong correlation between compensation and results, while HR demonstrates exceptional loyalty with 7.3 years tenure and competitive performance (3.80) despite having the lowest salary ($89,586). Engineering, though having the shortest tenure (4.5 years) and smallest team size (n=4), maintains competitive compensation ($91,848), potentially indicating either high turnover or recent hiring in a competitive market.

[Recommendations]

1. Rebalance compensation for high-tenure, lower-paid functions (especially Marketing and HR) by conducting a market benchmark and introducing targeted salary adjustments or retention bonuses to reduce the risk of losing experienced, loyal staff.

2. Deep-dive into Engineering\u2019s short tenure to determine whether it\u2019s driven by turnover or recent growth, then respond with clearer career paths, mentorship, and possibly sign-on/retention incentives if the market is highly competitive.

3. Analyze the specific practices in Sales that link higher pay to higher performance (e.g., variable pay, incentives, goal-setting) and selectively adapt those mechanisms to other departments, ensuring transparent performance metrics so increased compensation clearly ties to measurable results.

````
======================================================

/home/igor >

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

/home/igor > Sent to mshell (1483 bytes)

Received from GUI editor:

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

# Pattern 9: Routing  LLM Classifies - Conditional Branch Executes (Python)

# LLM classifies query \u2192 only the matching Python/LLM block runs

```python >query

print("Given a sorted array and a target value, what is the most efficient way to find if the target exists?")

```

<!--@1 <query >route

Classify this query into exactly one word: STATISTICS, ALGORITHM, or EXPLANATION.

Reply with only the single classification word.

-->

```python <query if=route:STATISTICS

import os, statistics

print("=== STATISTICS branch ===")

data = [2, 4, 4, 4, 5, 5, 7, 9]

print(f"Mean: {statistics.mean(data)}")

print(f"Median: {statistics.median(data)}")

print(f"Stdev: {statistics.stdev(data):.3f}")

```

```python <query if=route:ALGORITHM

import os

print("=== ALGORITHM branch ===")

def binary_search(arr, target):
````
````

```
    lo, hi = 0, len(arr) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if arr[mid] == target: return mid
        elif arr[mid] < target: lo = mid + 1
        else: hi = mid - 1
    return -1
arr = list(range(0, 100, 2))
for target in [42, 57, 88]:
    idx = binary_search(arr, target)
    print(f"Search for {target}: {'found at index ' + str(idx) if idx != -1 else 'not found'}")
print("Time complexity: O(log n)")
```

<!--@1 <query if=route:EXPLANATION

Answer this question in plain, accessible language without code. 2-3 sentences.

-->

```python <route
import os
print(f"\nQuery classified as: {open(os.environ['MSH_VAR_route']).read().strip()}")
```

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

Given a sorted array and a target value, what is the most efficient way to find if the target exists?

ALGORITHM

=== ALGORITHM branch ===

Search for 42: found at index 21

Search for 57: not found

Search for 88: found at index 44

Time complexity: O(log n)

/home/igor >

Query classified as: ALGORITHM

/home/igor >

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

/home/igor > Sent to mshell (1575 bytes)

Received from GUI editor:

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

# Pattern 10: Full Pipeline - All Patterns Combined (Python)

# Python generates primes \u2192 stats \u2192 async parallel LLMs \u2192 await \u2192 synthesis \u2192 formatted output

```python >raw_data
def sieve(n):
    is_prime = [True] * (n + 1)
    is_prime[0] = is_prime[1] = False
    for i in range(2, int(n**0.5) + 1):
        if is_prime[i]:
            for j in range(i*i, n+1, i):
                is_prime[j] = False
    return [i for i in range(2, n+1) if is_prime[i]]
primes = sieve(50)[:12]
print(' '.join(map(str, primes)))
```

```python <raw_data >stats
import os
nums = list(map(int, open(os.environ['MSH_VAR_raw_data']).read().strip().split()))
mean = sum(nums) / len(nums)
gaps = [nums[i+1] - nums[i] for i in range(len(nums)-1)]
```

````
avg_gap = sum(gaps) / len(gaps)

print(f"First {len(nums)} primes: {nums}")

print(f"Sum={sum(nums)} Mean={mean:.2f} Avg_gap={avg_gap:.2f}
Largest={max(nums)}")

```

<!--@1 <stats >analysis async

In one sentence, describe what is mathematically interesting about these prime numbers
and their distribution.

-->

<!--@2 <raw_data >poem async

Write a 2-line poem about prime numbers \u2014 their indivisible, solitary nature.

-->

```python await=analysis,poem

```

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

Combine these two perspectives into one elegant, memorable sentence.

-->

```python <combined

import os

combined = open(os.environ['MSH_VAR_combined']).read().strip()

width = max(60, len(combined) + 4)

border = '=' * width

print(border)

print(f"  {'PRIME NUMBERS \u2014 FINAL SYNTHESIS':^{width-2}}")

print(border)

print(combined)

print(border)

```

````
````

2 3 5 7 11 13 17 19 23 29 31 37

First 12 primes: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]

Sum=197 Mean=16.42 Avg_gap=3.18 Largest=37

[async llm] Launched PID 305849 \u2192 var=analysis (model @1)

[async llm] Launched PID 305850 \u2192 var=poem (model @2)

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

[async] Waiting for PID 305849 (var=analysis)...

[async] PID 305849 done (var=analysis)

The first 12 primes demonstrate the early irregularity of prime distribution, where despite an average gap of only 3.18 between consecutive primes, the gaps vary dramatically from 1 (between 2-3) to 6 (between 23-29), illustrating how prime density decreases and spacing becomes increasingly unpredictable even in this initial segment.

Among the first 12 primes, these lonely sentries stand with gaps ranging from 1 to 6 units apart\u2014solitary monarchs whose irregular spacing of 3.18 average steps reveals how indivisible numbers grow increasingly austere and unpredictable even in the crowded beginning of infinity's number line.

/home/igor >

==

PRIME NUMBERS \u2014 FINAL SYNTHESIS

==

Among the first 12 primes, these lonely sentries stand with gaps ranging from 1 to 6 units apart\u2014solitary monarchs whose irregular spacing of 3.18 average steps reveals how indivisible numbers grow increasingly austere and unpredictable even in the crowded beginning of infinity's number line.

==

/home/igor >

/home/igor > Sent to mshell (1298 bytes)

Received from GUI editor:

Pattern 11: Chained LLM Nodes - Python Style (Python)

Python sets inputs \u2192 LLM1 generates explanation \u2192 LLM2 extracts metadata \u2192 Python parses + displays

```python >topic

print("quantum computing")

```

```python >style

print("engaging and accessible for a non-technical audience")

```

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

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

Explain the topic in that style in exactly 3 vivid sentences with analogies.

-->

<!--@2 <explanation >metadata

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

KEYWORDS: word1, word2, word3, word4, word5

AUDIENCE: one word \u2014 beginner / intermediate / expert

READABILITY: one word \u2014 simple / moderate / complex

-->

```python <explanation <metadata

import os, re

explanation = open(os.environ['MSH_VAR_explanation']).read().strip()

metadata = open(os.environ['MSH_VAR_metadata']).read().strip()

kw = re.search(r'KEYWORDS:\s*(.+)', metadata)

aud = re.search(r'AUDIENCE:\s*(.+)', metadata)

read = re.search(r'READABILITY:\s*(.+)', metadata)

print("=== Explanation ===")

print(explanation)

print("\n=== Metadata ===")
```

```
print(f"Keywords   : {kw.group(1) if kw else 'n/a'}")

print(f"Audience   : {aud.group(1) if aud else 'n/a'}")

print(f"Readability: {read.group(1) if read else 'n/a'}")
```

----

quantum computing

engaging and accessible for a non-technical audience

Imagine a magical coin that can be heads, tails, and somehow both at the same time\u2014that's essentially how quantum computers process information, using particles that exist in multiple states simultaneously until you peek at them. While your regular computer is like a very fast librarian checking books one at a time, a quantum computer is like having a ghostly librarian who can read every book in multiple libraries at once, exploring countless possibilities in parallel. This spooky superpower allows quantum computers to crack certain puzzles\u2014like finding a needle in a haystack\u2014not by searching through every piece of hay, but by somehow knowing where all the needles are hiding before they even look.

KEYWORDS: quantum, computers, superposition, parallelism, puzzles

AUDIENCE: beginner

READABILITY: simple

/home/igor > === Explanation ===

Imagine a magical coin that can be heads, tails, and somehow both at the same time\u2014that's essentially how quantum computers process information, using particles that exist in multiple states simultaneously until you peek at them. While your regular computer is like a very fast librarian checking books one at a time, a quantum computer is like having a ghostly librarian who can read every book in multiple libraries at once, exploring countless possibilities in parallel. This spooky superpower allows quantum computers to crack certain puzzles\u2014like finding a needle in a haystack\u2014not by searching through every piece of hay, but by somehow knowing where all the needles are hiding before they even look.

=== Metadata ===

Keywords   : quantum, computers, superposition, parallelism, puzzles

Audience   : beginner

Readability: simple

/home/igor >

/home/igor > Sent to mshell (1557 bytes)

Received from GUI editor:

Pattern 12: Async Parallel 3 Models + Await Barrier + Synthesis (Python)

3 async LLM code reviews in parallel \u2192 Python await \u2192 LLM synthesizes \u2192 Python report

```python >code_snippet
code = '''
def find_user(users_list, user_id):
    for user in users_list:
        if user["id"] == user_id:
            return user["password"]
    return None
users = [{"id": i, "password": f"pass{i}"} for i in range(10000)]
result = find_user(users, 9999)
print(result)
'''
print(code.strip())
```

<!--@1 <code_snippet >review_perf async

Review this Python code focusing ONLY on performance and time complexity.

One paragraph with Big-O analysis and a concrete optimization suggestion.

-->

<!--@2 <code_snippet >review_sec async

Review this Python code focusing ONLY on security vulnerabilities.

One paragraph identifying specific risks and suggested fixes.

-->

<!--@3 <code_snippet >review_style async

Review this Python code focusing ONLY on Python style, PEP 8, type hints, and naming conventions.

One paragraph.

-->

```python await=review_perf,review_sec,review_style
```

<!--@1 <review_perf <review_sec <review_style >final_review

Synthesize these three code reviews into one comprehensive assessment.

Structure: VERDICT (one line), KEY ISSUES (3 bullet points), PRIORITY FIX (one sentence).

-->

```python <final_review
import os
review = open(os.environ['MSH_VAR_final_review']).read().strip()
print("=" * 42)
print("     ENSEMBLE CODE REVIEW REPORT")
print("=" * 42)
print(review)
print("\n[Generated from 3 independent model perspectives]")
```

```
def find_user(users_list, user_id):
    for user in users_list:
        if user["id"] == user_id:
            return user["password"]
    return None
users = [{"id": i, "password": f"pass{i}"} for i in range(10000)]
result = find_user(users, 9999)
```

print(result)

[async llm] Launched PID 314475 \u2192 var=review_perf (model @1)

[async llm] Launched PID 314477 \u2192 var=review_sec (model @2)

[async llm] Launched PID 314480 \u2192 var=review_style (model @3)

[async] await= barrier: waiting for vars: review_perf,review_sec,review_style

[async] Waiting for PID 314475 (var=review_perf)...

[async] PID 314475 done (var=review_perf)

This code implements a linear search with O(n) time complexity, where n is the length of the users list, making it inefficient for large datasets as it potentially scans every user to find a match. In the worst case (searching for user_id 9999 in a 10,000-user list), it performs 10,000 comparisons, and average case still requires n/2 iterations. The most effective optimization would be to convert the users list to a dictionary with user IDs as keys: `users_dict = {user["id"]: user["password"] for user in users_list}`, then use `users_dict.get(user_id)` for O(1) average-case lookups, reducing search time from potentially 10,000 operations to just 1, though this requires O(n) space and one-time O(n) preprocessing.

VERDICT: This code has critical security vulnerabilities and poor performance characteristics that make it unsuitable for production use.

KEY ISSUES:

\u2022 **Security Risk**: Stores and returns plaintext passwords, creating a direct credential exposure vulnerability if data is leaked or logged

\u2022 **Performance Problem**: Uses inefficient O(n) linear search that scales poorly with large user datasets, potentially requiring thousands of comparisons

\u2022 **Design Flaw**: Violates security best practices by making sensitive authentication data retrievable and printable

PRIORITY FIX: Immediately replace plaintext password storage with hashed passwords using bcrypt or argon2, convert the user list to a dictionary for O(1) lookups, and modify the function to never return or display password data.

/home/igor > ==

ENSEMBLE CODE REVIEW REPORT

==

VERDICT: This code has critical security vulnerabilities and poor performance characteristics that make it unsuitable for production use.

KEY ISSUES:

\u2022 **Security Risk**: Stores and returns plaintext passwords, creating a direct credential exposure vulnerability if data is leaked or logged

\u2022 **Performance Problem**: Uses inefficient O(n) linear search that scales poorly with large user datasets, potentially requiring thousands of comparisons

\u2022 **Design Flaw**: Violates security best practices by making sensitive authentication data retrievable and printable

PRIORITY FIX: Immediately replace plaintext password storage with hashed passwords using bcrypt or argon2, convert the user list to a dictionary for O(1) lookups, and modify the function to never return or display password data.

[Generated from 3 independent model perspectives]

/home/igor >

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

Received from GUI editor:

Pattern 13: WHILE Loop - Iterative Convergence with LLM Commentary (Python)

Computes Fibonacci numbers iteratively; LLM comments on each; exits after 5 iterations

```python >status

print("running")

```

```python >iteration

print("0")

```

```python >fib_val

print("1")

```

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

```python <iteration <status >iteration >status >fib_val

import os
```

```
val = int(open(os.environ['MSH_VAR_iteration']).read().strip())
val += 1
a, b = 1, 1
for _ in range(val - 1):
    a, b = b, a + b
fib = a
with open(os.environ['MSH_VAR_iteration'], 'w') as f: f.write(str(val))
with open(os.environ['MSH_VAR_fib_val'], 'w') as f: f.write(str(fib))
with open(os.environ['MSH_VAR_status'], 'w') as f: f.write("done" if val >= 5 else "running")
```

<!--@1 <fib_val >fact

The input contains a number that is part of the Fibonacci sequence. In one sentence, share something mathematically interesting about this specific number.

-->

```python <fact
import os
itr = open(os.environ['MSH_VAR_iteration']).read().strip()
fib = open(os.environ['MSH_VAR_fib_val']).read().strip()
fact = open(os.environ['MSH_VAR_fact']).read().strip()
print(f"[iter {itr}] Fib={fib} -> {fact}")
```

<!--@end_while-->

```python <iteration
import os
print(f"=== WHILE done. Completed {open(os.environ['MSH_VAR_iteration']).read().strip()} iterations. ===")
```

----

running

0

1

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

The number 1 is mathematically unique in the Fibonacci sequence as it's the only value that appears twice (at positions F\u2081 = 1 and F\u2082 = 1), making it the foundational element that enables the recursive definition F(n) = F(n-1) + F(n-2) for all subsequent terms.

[iter 1] Fib=1 -> The number 1 is mathematically unique in the Fibonacci sequence as it's the only value that appears twice (at positions F\u2081 = 1 and F\u2082 = 1), making it the foundational element that enables the recursive definition F(n) = F(n-1) + F(n-2) for all subsequent terms.

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

The number 1 is the only positive integer that is simultaneously its own square ($1^2 = 1$), its own factorial (1! = 1), and its own power raised to any exponent (1\u207f = 1), making it a mathematical fixed point for multiple fundamental operations.

[iter 2] Fib=1 -> The number 1 is the only positive integer that is simultaneously its own square ($1^2 = 1$), its own factorial (1! = 1), and its own power raised to any exponent (1\u207f = 1), making it a mathematical fixed point for multiple fundamental operations.

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

The number 2 is the only even prime number and the smallest prime in the Fibonacci sequence, making it uniquely both a fundamental building block of all even numbers through multiplication and an indivisible element that cannot be factored further.

[iter 3] Fib=2 -> The number 2 is the only even prime number and the smallest prime in the Fibonacci sequence, making it uniquely both a fundamental building block of all even numbers through multiplication and an indivisible element that cannot be factored further.

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

The number 3 is the smallest odd prime and forms the first primitive Pythagorean triple (3, 4, 5), making it simultaneously a prime building block of arithmetic, a Fibonacci term, and the leg of the most fundamental right triangle with integer sides.

[iter 4] Fib=3 -> The number 3 is the smallest odd prime and forms the first primitive Pythagorean triple (3, 4, 5), making it simultaneously a prime building block of arithmetic, a Fibonacci term, and the leg of the most fundamental right triangle with integer sides.

[while] iteration 5 \u2014 condition met, executing body

The number 5 is remarkably one of only two Fibonacci numbers that equals its own position in the sequence (F\u2085 = 5, along with F\u2081 = 1), and it's also the only

prime Fibonacci number that equals the count of Platonic solids in three-dimensional geometry.

[iter 5] Fib=5 -> The number 5 is remarkably one of only two Fibonacci numbers that equals its own position in the sequence (F\u2085 = 5, along with F\u2081 = 1), and it's also the only prime Fibonacci number that equals the count of Platonic solids in three-dimensional geometry.

[while] condition 'status:running' not met, exiting after 5 iter

/home/igor > === WHILE done. Completed 5 iterations. ===

/home/igor >

/home/igor > Sent to mshell (733 bytes)

Received from GUI editor:

Pattern 14: FOREACH - LLM Processes Each Item in a List (Python)

Python creates a newline-separated list of libraries; FOREACH iterates; LLM describes each

```python >topics

print("pandas\nnumpy\nscikit-learn\nmatplotlib\nfastapi")

```

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

<!--@1 <topic >description

The input contains a Python library name. In one sentence, describe what it is best used for and name one iconic real-world use case.

-->

```python <description

import os

topic = open(os.environ['MSH_VAR_topic']).read().strip()

desc = open(os.environ['MSH_VAR_description']).read().strip()

print(f"> {topic}")

print(f"  {desc}")
```

print()

```

&lt;!--@end_foreach--&gt;

```python

print("=== All 5 Python libraries described. ===")

```

----

pandas

numpy

scikit-learn

matplotlib

fastapi

[foreach] iter 1: topic=pandas

Pandas is best used for data manipulation, analysis, and cleaning of structured data through its powerful DataFrame and Series objects, with iconic real-world applications including financial time series analysis where institutions like Goldman Sachs use it to process millions of stock price records, calculate moving averages, and generate trading signals.

> pandas

Pandas is best used for data manipulation, analysis, and cleaning of structured data through its powerful DataFrame and Series objects, with iconic real-world applications including financial time series analysis where institutions like Goldman Sachs use it to process millions of stock price records, calculate moving averages, and generate trading signals.

[foreach] iter 2: topic=numpy

NumPy is best used for high-performance numerical computing and mathematical operations on large multi-dimensional arrays, with iconic real-world applications including NASA's image processing pipelines where it handles massive satellite imagery datasets and performs complex mathematical transformations for space mission analysis.

> numpy

NumPy is best used for high-performance numerical computing and mathematical operations on large multi-dimensional arrays, with iconic real-world applications including
```

NASA's image processing pipelines where it handles massive satellite imagery datasets and performs complex mathematical transformations for space mission analysis.

[foreach] iter 3: topic=scikit-learn

Scikit-learn is best used for machine learning tasks including classification, regression, clustering, and model evaluation through its comprehensive suite of algorithms and tools, with iconic real-world applications including Netflix's recommendation system where it processes millions of user ratings to predict movie preferences and suggest personalized content.

> scikit-learn

Scikit-learn is best used for machine learning tasks including classification, regression, clustering, and model evaluation through its comprehensive suite of algorithms and tools, with iconic real-world applications including Netflix's recommendation system where it processes millions of user ratings to predict movie preferences and suggest personalized content.

[foreach] iter 4: topic=matplotlib

Matplotlib is best used for creating static, animated, and interactive data visualizations including plots, charts, and graphs in Python, with iconic real-world applications including the Event Horizon Telescope project where scientists used it to generate the groundbreaking visualizations of black holes from radio telescope data.

> matplotlib

Matplotlib is best used for creating static, animated, and interactive data visualizations including plots, charts, and graphs in Python, with iconic real-world applications including the Event Horizon Telescope project where scientists used it to generate the groundbreaking visualizations of black holes from radio telescope data.

[foreach] iter 5: topic=fastapi

FastAPI is best used for building high-performance REST APIs and web services with automatic OpenAPI documentation generation and built-in async support, with iconic real-world applications including Uber's microservices architecture where it handles millions of ride requests per day with fast response times and automatic API documentation for their engineering teams.

> fastapi

FastAPI is best used for building high-performance REST APIs and web services with automatic OpenAPI documentation generation and built-in async support, with iconic real-world applications including Uber's microservices architecture where it handles millions of ride requests per day with fast response times and automatic API documentation for their engineering teams.

/home/igor > === All 5 Python libraries described. ===

/home/igor >

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

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

Received from GUI editor:

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

# Pattern 15: TRY/CATCH - Safe Execution with Error Capture (Python)

\# TRY attempts json.loads on malformed JSON \u2192 fails \u2192 CATCH handles \u2192 safe regex fallback continues

```python >raw_json
print('{"name": "Alice", "age": 30, "city": broken_value}')
```

<!--@try-->

```python <raw_json >parsed
import os, json
raw = open(os.environ['MSH_VAR_raw_json']).read().strip()
data = json.loads(raw)
print(data)
```

<!--@catch >error-->

```python
print("=== Caught error: try_block_failed ===")
print("Malformed JSON detected. Switching to safe fallback parser.")
```

<!--@end_try-->

```python <raw_json >safe_result
import os, re
raw = open(os.environ['MSH_VAR_raw_json']).read().strip()
fields = re.findall(r'"(\w+)":\s*(?:"([^"]*)"|\d+|(\w+))', raw)
```

```
result = {k: v1 or v2 for k, v1, v2 in fields}
print(f"Safe extraction: {result}")
print(f"Recovered fields: {list(result.keys())}")
```

```python <safe_result
import os
print("=== Safe Result ===")
print(open(os.environ['MSH_VAR_safe_result']).read().strip())
```

```
{"name": "Alice", "age": 30, "city": broken_value}
[try] executing try block
Traceback (most recent call last):
  File "/tmp/mshell_script_301141.py", line 3, in <module>
    data = json.loads(raw)
           ^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/json/__init__.py", line 346, in loads
    return _default_decoder.decode(s)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/json/decoder.py", line 337, in decode
    obj, end = self.raw_decode(s, idx=_w(s, 0).end())
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.12/json/decoder.py", line 355, in raw_decode
    raise JSONDecodeError("Expecting value", s, err.value) from None
json.decoder.JSONDecodeError: Expecting value: line 1 column 38 (char 37)
[try] try block failed, executing catch block
=== Caught error: try_block_failed ===
Malformed JSON detected. Switching to safe fallback parser.
```

Safe extraction: {'name': 'Alice', 'age': '', 'city': 'broken_value'}

Recovered fields: ['name', 'age', 'city']

/home/igor > === Safe Result ===

Safe extraction: {'name': 'Alice', 'age': '', 'city': 'broken_value'}

Recovered fields: ['name', 'age', 'city']

/home/igor >

/home/igor > Sent to mshell (1417 bytes)

Received from GUI editor:

Pattern 16: SPLIT + MERGE - Divide-and-Conquer Analysis (Python)

Python creates 2-series dataset \u2192 SPLIT \u2192 async parallel LLM analysis \u2192 MERGE \u2192 synthesis

```python >dataset
print("climate:temp_anomaly:+1.2,+0.8,+1.5,+2.1,+1.9\nclimate:co2_ppm:315,325,340,370,415")
```

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

<!--@1 <dataset_1 >analysis1 async

The input contains a labeled time-series dataset. In one sentence, describe the trend, magnitude of change, and what it suggests scientifically.

-->

<!--@2 <dataset_2 >analysis2 async

The input contains a labeled time-series dataset. In one sentence, describe the trend, magnitude of change, and what it suggests scientifically.

-->

```python await=analysis1,analysis2
```

<!--@merge-->

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

Combine these two time-series analyses into one unified scientific summary. Connect them causally if appropriate. Two sentences max.

-->

```python <combined

import os

d1 = open(os.environ['MSH_VAR_dataset_1']).read().strip()

d2 = open(os.environ['MSH_VAR_dataset_2']).read().strip()

a1 = open(os.environ['MSH_VAR_analysis1']).read().strip()

a2 = open(os.environ['MSH_VAR_analysis2']).read().strip()

combined = open(os.environ['MSH_VAR_combined']).read().strip()

print(f"=== Series 1: {d1} ===")

print(f"Analysis: {a1}\n")

print(f"=== Series 2: {d2} ===")

print(f"Analysis: {a2}\n")

print(f"=== Merged Insight ===")

print(combined)
```

----

climate:temp_anomaly:+1.2,+0.8,+1.5,+2.1,+1.9

climate:co2_ppm:315,325,340,370,415

[split] dataset_1 = climate:temp_anomaly:+1.2,+0.8,+1.5,+2.1,+1.9

[split] dataset_2 = climate:co2_ppm:315,325,340,370,415

[async llm] Launched PID 314779 \u2192 var=analysis1 (model @1)

[async llm] Launched PID 314780 \u2192 var=analysis2 (model @2)

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

[async] Waiting for PID 314779 (var=analysis1)...

[async] PID 314779 done (var=analysis1)

The temperature anomaly data shows a persistent warming trend with values ranging from +0.8°C to +2.1°C above baseline, indicating consistent above-average temperatures with a

notable spike to +2.1°C, which scientifically suggests accelerating climate change with potential implications for extreme weather events and ecosystem disruption.

The CO\u2082 concentration increase from 315 to 415 ppm directly correlates with the persistent temperature anomalies of +0.8°C to +2.1°C above baseline, demonstrating the causal relationship between anthropogenic greenhouse gas accumulation and accelerating global warming. This 100 ppm rise in atmospheric CO\u2082 is driving the observed temperature spikes and represents a significant climate forcing mechanism that threatens to trigger more extreme weather events and widespread ecosystem disruption.

/home/igor > === Series 1: climate:temp_anomaly:+1.2,+0.8,+1.5,+2.1,+1.9 ===

Analysis: The temperature anomaly data shows a persistent warming trend with values ranging from +0.8°C to +2.1°C above baseline, indicating consistent above-average temperatures with a notable spike to +2.1°C, which scientifically suggests accelerating climate change with potential implications for extreme weather events and ecosystem disruption.

=== Series 2: climate:co2_ppm:315,325,340,370,415 ===

Analysis: The CO\u2082 concentration rises steadily from 315 to 415 ppm, an increase of about 100 ppm, indicating a strong upward trend consistent with significant anthropogenic greenhouse gas accumulation and ongoing climate forcing.

=== Merged Insight ===

The CO\u2082 concentration increase from 315 to 415 ppm directly correlates with the persistent temperature anomalies of +0.8°C to +2.1°C above baseline, demonstrating the causal relationship between anthropogenic greenhouse gas accumulation and accelerating global warming. This 100 ppm rise in atmospheric CO\u2082 is driving the observed temperature spikes and represents a significant climate forcing mechanism that threatens to trigger more extreme weather events and widespread ecosystem disruption.

/home/igor >

/home/igor > Sent to mshell (1340 bytes)

Received from GUI editor:

Pattern 17: CONFIG Node - Parameterized Pipeline (Python)

CONFIG documents parameters; Python initializes runtime vars; LLMs generate + extract; Python formats report

```config

domain=distributed systems
```

complexity=intermediate

output_format=structured explanation + keywords

```

```python &gt;domain

print("distributed systems")

```

```python &gt;complexity

print("intermediate")

```

<!--@1 <domain <complexity >explanation

The first input is a technical domain. The second input is the complexity level of the target audience.

Explain the core concepts of this domain for that audience in exactly 4 concrete sentences.

-->

<!--@2 <explanation >keywords

The input is a technical explanation. Extract exactly 5 keywords that best represent the core concepts.

Reply with only the comma-separated list, nothing else.

-->

```python <explanation <keywords >report

import os

domain = open(os.environ['MSH_VAR_domain']).read().strip()

complexity = open(os.environ['MSH_VAR_complexity']).read().strip()

explanation = open(os.environ['MSH_VAR_explanation']).read().strip()

keywords = open(os.environ['MSH_VAR_keywords']).read().strip()

print('\n'.join([

    "=== Workflow Report ===",

    f"Domain     : {domain}",

    f"Level      : {complexity}",

```
    "",
    "Explanation:",
    explanation,
    "",
    f"Keywords   : {keywords}",
]))
```

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

[config] domain=distributed systems

[config] complexity=intermediate

[config] output_format=structured explanation + keywords

distributed systems

intermediate

Distributed systems are networks of independent computers that work together to appear as a single coherent system to users, like how Netflix serves millions of users simultaneously by coordinating thousands of servers across multiple data centers. The fundamental challenge is achieving consistency across nodes while maintaining availability and partition tolerance\u2014known as the CAP theorem\u2014which forces architects to make trade-offs since you can only guarantee two of these three properties simultaneously. Key concepts include consensus algorithms like Raft for leader election, eventual consistency for data synchronization, and fault tolerance through replication, where systems continue operating even when individual nodes fail. Common architectural patterns include load balancing to distribute traffic, horizontal scaling to add more machines rather than upgrading existing ones, and message queues like Apache Kafka to handle asynchronous communication between services.

distributed systems,CAP theorem,consensus,eventual consistency,fault tolerance

=== Workflow Report ===

Domain     : distributed systems

Level      : intermediate

Explanation:

Distributed systems are networks of independent computers that work together to appear as a single coherent system to users, like how Netflix serves millions of users simultaneously by coordinating thousands of servers across multiple data centers. The

fundamental challenge is achieving consistency across nodes while maintaining availability and partition tolerance\u2014known as the CAP theorem\u2014which forces architects to make trade-offs since you can only guarantee two of these three properties simultaneously. Key concepts include consensus algorithms like Raft for leader election, eventual consistency for data synchronization, and fault tolerance through replication, where systems continue operating even when individual nodes fail. Common architectural patterns include load balancing to distribute traffic, horizontal scaling to add more machines rather than upgrading existing ones, and message queues like Apache Kafka to handle asynchronous communication between services.

Keywords : distributed systems,CAP theorem,consensus,eventual consistency,fault tolerance

/home/igor >

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

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

Received from GUI editor:

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

# mshell C Language Edition

---

### Pattern 18 FOREACH + Async LLM: Parallel C Algorithm Batch Analysis

```c >algorithms
#include <stdio.h>

int main(void) { printf("dijkstra\naes_encryption\nlru_cache\n"); return 0; }
```

<!--@foreach algo in algorithms-->

<!--@1 <algo >explanation async

The input is a C algorithm/data structure name.

Explain in one sentence for a beginner C programmer.

-->

<!--@2 <algo >usecase async

The input is a C algorithm/data structure name.

Name one real-world C system that uses it and why. One sentence.

-->

```c await=explanation,usecase
```

```c <explanation <usecase
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    printf("=== ");
    FILE *f = fopen(getenv("MSH_VAR_algo"), "r"); int c;
    while ((c = fgetc(f)) != EOF && c != '\n') putchar(c); fclose(f); printf(" ===\n");
    printf("Explanation : ");
    f = fopen(getenv("MSH_VAR_explanation"), "r");
    while ((c = fgetc(f)) != EOF) putchar(c); fclose(f);
    printf("Real-world  : ");
    f = fopen(getenv("MSH_VAR_usecase"), "r");
    while ((c = fgetc(f)) != EOF) putchar(c); fclose(f); printf("\n");
    return 0;
}
```

<!--@end_foreach-->

```c
#include <stdio.h>
int main(void) { printf("=== All algorithms analyzed ===\n"); return 0; }
```

[DEBUG compile_c] libs: [-lm]

[DEBUG compile_c] cmd: [gcc -Wall -Wextra -std=c11 -o /tmp/mshell_validate_301141.out /tmp/mshell_validate_301141.c -lm 2>&1]

dijkstra

aes_encryption

lru_cache

[foreach] iter 1: algo=dijkstra

[async llm] Launched PID 314890 \u2192 var=explanation (model @1)

[async llm] Launched PID 314891 \u2192 var=usecase (model @2)

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

[async] Waiting for PID 314890 (var=explanation)...

[async] PID 314890 done (var=explanation)

Dijkstra's algorithm is a graph search method that finds the shortest path between nodes by maintaining an array of distances and repeatedly selecting the unvisited node with the smallest known distance, making it useful for problems like finding the shortest route in a road network represented as a weighted graph in C.

[DEBUG compile_c] libs: [-lm]

[DEBUG compile_c] cmd: [gcc -Wall -Wextra -std=c11 -o /tmp/mshell_validate_301141.out /tmp/mshell_validate_301141.c -lm 2>&1]

=== dijkstra ===

Explanation : Dijkstra's algorithm is a graph search method that finds the shortest path between nodes by maintaining an array of distances and repeatedly selecting the unvisited node with the smallest known distance, making it useful for problems like finding the shortest route in a road network represented as a weighted graph in C.

Real-world : The CPython interpreter (written in C) implements an LRU-style cache for method lookups and other hotspots to avoid repeated expensive computations and significantly speed up frequently executed operations.

[foreach] iter 2: algo=aes_encryption

[async llm] Launched PID 314928 \u2192 var=explanation (model @1)

[async llm] Launched PID 314929 \u2192 var=usecase (model @2)

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

[async] Waiting for PID 314928 (var=explanation)...

[async] PID 314928 done (var=explanation)

AES encryption is a symmetric cryptographic algorithm implemented in C using fixed-size arrays and bitwise operations to transform plaintext data into ciphertext using the same

key for both encryption and decryption, requiring careful handling of 128-bit blocks and key scheduling through nested loops and lookup tables.

[DEBUG compile_c] libs: [-lm]

[DEBUG compile_c] cmd: [gcc -Wall -Wextra -std=c11 -o /tmp/mshell_validate_301141.out /tmp/mshell_validate_301141.c -lm 2>&1]

=== aes_encryption ===

Explanation : AES encryption is a symmetric cryptographic algorithm implemented in C using fixed-size arrays and bitwise operations to transform plaintext data into ciphertext using the same key for both encryption and decryption, requiring careful handling of 128-bit blocks and key scheduling through nested loops and lookup tables.

Real-world : OpenSSL uses AES encryption in C to provide secure communication for protocols like TLS/SSL by efficiently encrypting and decrypting network traffic.

[foreach] iter 3: algo=lru_cache

[async llm] Launched PID 314966 \u2192 var=explanation (model @1)

[async llm] Launched PID 314967 \u2192 var=usecase (model @2)

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

[async] Waiting for PID 314966 (var=explanation)...

[async] PID 314966 done (var=explanation)

An LRU (Least Recently Used) cache is a data structure that stores a fixed number of key-value pairs and automatically removes the oldest unused item when full, typically implemented in C using a hash table for fast lookups combined with a doubly-linked list to track which items were accessed most recently.

[DEBUG compile_c] libs: [-lm]

[DEBUG compile_c] cmd: [gcc -Wall -Wextra -std=c11 -o /tmp/mshell_validate_301141.out /tmp/mshell_validate_301141.c -lm 2>&1]

=== lru_cache ===

Explanation : An LRU (Least Recently Used) cache is a data structure that stores a fixed number of key-value pairs and automatically removes the oldest unused item when full, typically implemented in C using a hash table for fast lookups combined with a doubly-linked list to track which items were accessed most recently.

Real-world : The Linux kernel uses LRU-based page caches to keep recently accessed disk pages in memory, improving performance by quickly serving repeated reads without hitting slower storage.

[DEBUG compile_c] libs: [-lm]

[DEBUG compile_c] cmd: [gcc -Wall -Wextra -std=c11 -o /tmp/mshell_validate_301141.out /tmp/mshell_validate_301141.c -lm 2>&1]

/home/igor > === All algorithms analyzed ===

/home/igor >

/home/igor > Sent to mshell (1459 bytes)

Received from GUI editor:

Pattern 19: WHILE Quality Gate - Generate Until Threshold (Python)

LLM generates Python function; LLM scores it 1-10; loop exits when score >= 8

```python >task

print("Write a Python function that reads a CSV file and returns the top N rows sorted by a specified column, handling missing values gracefully.")

```

```python >status

print("running")

```

```python >iteration

print("0")

```

```python >score

print("0")

```

```python >snippet

print("")

```

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

```python <iteration >iteration

import os
```

```
val = int(open(os.environ['MSH_VAR_iteration']).read().strip())

print(val + 1)
```

<!--@1 <task >snippet

Write a clean, idiomatic Python function that solves this task. Return ONLY the code, no markdown fences, no explanation.

-->

<!--@2 <snippet >score

Rate this Python function 1-10 based on: correctness, idiomatic style, error handling, readability.

Reply with ONLY the integer score, nothing else.

-->

```python <iteration <score <snippet >status
import os

itr = open(os.environ['MSH_VAR_iteration']).read().strip()

sc = open(os.environ['MSH_VAR_score']).read().strip()

print(f"[Iter {itr}] Score={sc}")

try:
    print("done" if int(sc) >= 8 else "running")
except ValueError:
    print("running")
```

<!--@end_while-->

```python <snippet <score
import os

print(f"=== Accepted snippet (score={open(os.environ['MSH_VAR_score']).read().strip()}) ===")

print(open(os.environ['MSH_VAR_snippet']).read().strip())
```

Write a Python function that reads a CSV file and returns the top N rows sorted by a specified column, handling missing values gracefully.

running

0

0

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

1

```
import pandas as pd

def get_top_n_rows(file_path, sort_column, n=10, ascending=False):
  try:
    df = pd.read_csv(file_path)
    if sort_column not in df.columns:
      raise ValueError(f"Column '{sort_column}' not found in the CSV file")
    df_cleaned = df.dropna(subset=[sort_column])
    df_sorted = df_cleaned.sort_values(by=sort_column, ascending=ascending)
    return df_sorted.head(n)
  except FileNotFoundError:
    raise FileNotFoundError(f"CSV file '{file_path}' not found")
  except pd.errors.EmptyDataError:
    raise ValueError("The CSV file is empty")
  except Exception as e:
    raise Exception(f"Error processing CSV file: {str(e)}")
```

9

[Iter 1] Score=9

done

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

/home/igor > === Accepted snippet (score=9) ===

```python
import pandas as pd

def get_top_n_rows(file_path, sort_column, n=10, ascending=False):
    try:
        df = pd.read_csv(file_path)
        if sort_column not in df.columns:
            raise ValueError(f"Column '{sort_column}' not found in the CSV file")
        df_cleaned = df.dropna(subset=[sort_column])
        df_sorted = df_cleaned.sort_values(by=sort_column, ascending=ascending)
        return df_sorted.head(n)
    except FileNotFoundError:
        raise FileNotFoundError(f"CSV file '{file_path}' not found")
    except pd.errors.EmptyDataError:
        raise ValueError("The CSV file is empty")
    except Exception as e:
        raise Exception(f"Error processing CSV file: {str(e)}")
```

/home/igor >

Sent to mshell (2342 bytes)

Received from GUI editor:

Pattern 20: SPLIT + Async + MERGE Map-Reduce Pipeline (Python)

Python text \u2192 3 chunks \u2192 async MAP (3x LLM concept extraction) \u2192 await \u2192 MERGE \u2192 REDUCE synthesis

```
```python >raw_text
text = (
    "Python's data model allows objects to implement special methods that define behavior for built-in operations. "
    "By implementing __len__, __getitem__, and __iter__, a class can behave like a sequence. "
```
```

"Generators and coroutines use yield to produce lazy sequences and enable cooperative multitasking. "

"The asyncio framework builds on coroutines to provide concurrent I/O-bound execution on a single thread. "

"Context managers implement __enter__ and __exit__ to guarantee resource cleanup with the with statement."

)

print(text)

```

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

```python &lt;raw_text &gt;chunk1

import os

sents = open(os.environ['MSH_VAR_raw_text']).read().strip().split('. ')

sents = [s.strip() for s in sents if s.strip()]

print(sents[0] if sents else '')

```

```python &lt;raw_text &gt;chunk2

import os

sents = open(os.environ['MSH_VAR_raw_text']).read().strip().split('. ')

sents = [s.strip() for s in sents if s.strip()]

print('. '.join(sents[1:3]) if len(sents) > 1 else '')

```

```python &lt;raw_text &gt;chunk3

import os

sents = open(os.environ['MSH_VAR_raw_text']).read().strip().split('. ')

sents = [s.strip() for s in sents if s.strip()]

print('. '.join(sents[3:]) if len(sents) > 3 else '')

```

<!--@1 <chunk1 >concept1 async
```

State the main Python concept in this text in 3 words max.

-->

<!--@1 <chunk2 >concept2 async

State the main Python concept in this text in 3 words max.

-->

<!--@1 <chunk3 >concept3 async

State the main Python concept in this text in 3 words max.

-->

```python await=concept1,concept2,concept3
```

<!--@merge-->

<!--@2 <concept1 <concept2 <concept3 >theme

These are three Python concept labels. Synthesize them into one sentence describing the unifying Python philosophy they represent.

-->

```python <concept1 <concept2 <concept3 <theme
import os

print("=== Map ===")

print(f"Chunk 1: {open(os.environ['MSH_VAR_concept1']).read().strip()}")

print(f"Chunk 2: {open(os.environ['MSH_VAR_concept2']).read().strip()}")

print(f"Chunk 3: {open(os.environ['MSH_VAR_concept3']).read().strip()}")

print("\n=== Reduce ===")

print(open(os.environ['MSH_VAR_theme']).read().strip())
```

Python's data model allows objects to implement special methods that define behavior for built-in operations. By implementing __len__, __getitem__, and __iter__, a class can behave like a sequence. Generators and coroutines use yield to produce lazy sequences and enable cooperative multitasking. The asyncio framework builds on coroutines to provide

concurrent I/O-bound execution on a single thread. Context managers implement __enter__ and __exit__ to guarantee resource cleanup with the with statement.

[split] raw_text_1 = Python's data model allows objects to implement special methods that define behavior for built-in operations. By implementing __len__, __getitem__, and __iter__, a class can behave like a sequence. Generators and coroutines use yield to produce lazy sequences and enable cooperative multitasking. The asyncio framework builds on coroutines to provide concurrent I/O-bound execution on a single thread. Context managers implement __enter__ and __exit__ to guarantee resource cleanup with the with statement.

Python's data model allows objects to implement special methods that define behavior for built-in operations

By implementing __len__, __getitem__, and __iter__, a class can behave like a sequence. Generators and coroutines use yield to produce lazy sequences and enable cooperative multitasking

The asyncio framework builds on coroutines to provide concurrent I/O-bound execution on a single thread. Context managers implement __enter__ and __exit__ to guarantee resource cleanup with the with statement.

[async llm] Launched PID 315188 \u2192 var=concept1 (model @1)

[async llm] Launched PID 315190 \u2192 var=concept2 (model @1)

[async llm] Launched PID 315192 \u2192 var=concept3 (model @1)

[async] await= barrier: waiting for vars: concept1,concept2,concept3

[async] Waiting for PID 315188 (var=concept1)...

[async] PID 315188 done (var=concept1)

Special methods implementation

They embody Python\u2019s philosophy of providing rich, explicit protocols (like special methods and the sequence protocol) and structured concurrency (via asyncio coroutines) so that objects, collections, and asynchronous tasks all integrate cleanly into a uniform, extensible programming model.

/home/igor > === Map ===

Chunk 1: Special methods implementation

Chunk 2: **Sequence Protocol Implementation**

Chunk 3: Asyncio coroutines concurrency

=== Reduce ===

They embody Python\u2019s philosophy of providing rich, explicit protocols (like special methods and the sequence protocol) and structured concurrency (via asyncio coroutines) so that objects, collections, and asynchronous tasks all integrate cleanly into a uniform, extensible programming model.

/home/igor >

/home/igor > Sent to mshell (2261 bytes)

Received from GUI editor:

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

What it does: LLM generates Python code for a math task. TRY executes it via subprocess. On success writes `ok`. On failure CATCH writes `fail`. Next iteration LLM sees the previous error and self-corrects. Loop exits when `result == ok`.

Key patterns:

- CATCH block does not use `<last_error` \u2014 writes `fail` to `>result` directly.

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

- CATCH declares `>last_error` so the parser registers the variable.

Flow diagram:

```

python >task, >result="fail", >last_error="none"

[LOOP max=3 until=result:ok]

 @1 <task <last_error >code      (generate / fix code)

 python <code                  (print)

 [TRY]

  python <code >result         (execute; writes "ok" on success)

 [CATCH >last_error]

  python >result               (writes "fail")

 [END_TRY]

[END_LOOP]

python <result  (final status)
```

````
```

**Code:**

```python >task

print('Write Python code that loads JSON \'{"scores": [95, 87, 92, 78, 88]}\', computes mean and standard deviation without external libraries, and prints: mean=XX.X std=X.X. Use only the math module.')

```

```python >result

print("fail")

```

```python >last_error

print("none")

```

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

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

The first input is a Python coding task. The second input is the error from the previous attempt (or "none").

Return ONLY Python code that solves the task \u2014 no markdown fences, no explanation.

If there was a previous error, fix it.

-->

```python <code

import os

print("=== Generated Code ===")

print(open(os.environ['MSH_VAR_code']).read())

```

<!--@try-->

```python <code >result

import os, subprocess, sys

path = os.environ['MSH_VAR_code']
````

```
result = subprocess.run([sys.executable, path], capture_output=True, text=True)
if result.returncode != 0:
    raise RuntimeError(result.stderr)
print(result.stdout.strip())
print("ok")
```

<!--@catch >last_error-->

```python >result
print("fail")
```

<!--@end_try-->

<!--@end_loop-->

```python <result
import os
print(f"=== Final status: {open(os.environ['MSH_VAR_result']).read().strip()} ===")
```

Write Python code that loads JSON '{"scores": [95, 87, 92, 78, 88]}', computes mean and standard deviation without external libraries, and prints: mean=XX.X std=X.X. Use only the math module.

fail

none

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

import json

import math

json_data = '{"scores": [95, 87, 92, 78, 88]}'

data = json.loads(json_data)

scores = data["scores"]

mean = sum(scores) / len(scores)

variance = sum((x - mean) ** 2 for x in scores) / len(scores)

std = math.sqrt(variance)

print(f"mean={mean:.1f} std={std:.1f}")

[try] executing try block

=== Generated Code ===

import json

import math

json_data = '{"scores": [95, 87, 92, 78, 88]}'

data = json.loads(json_data)

scores = data["scores"]

mean = sum(scores) / len(scores)

variance = sum((x - mean) ** 2 for x in scores) / len(scores)

std = math.sqrt(variance)

print(f"mean={mean:.1f} std={std:.1f}")

mean=88.0 std=5.8

ok

[try] try block succeeded

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

/home/igor > === Final status: mean=88.0 std=5.8

ok ===

/home/igor >

/home/igor > Sent to mshell (2672 bytes)

Received from GUI editor:

Pattern 22 Multi-Variable Output: Structured Field Extraction

What it does: LLM responds in a strict 3-line format to a Python library description. Python parses the response and writes three structured fields to separate variables

(`summary`, `tags`, `difficulty`) directly via `MSH_VAR_*` file writes. LLM @2 generates a learning recommendation. Python prints the complete output.

Key patterns:

- Multiple `>outvar` on a CODE block: block runs without stdout capture \u2014 must write via `open(os.environ['MSH_VAR_*'], 'w')`.

- Multiple `>outvar` on an LLM directive: full response is copied identically into each variable.

- The `MSH_VAR_*` environment variables are pre-set by the parser.

Flow diagram:

```

python >input

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

python <raw_response >summary >tags >difficulty

  (parse with regex; write via open(os.environ['MSH_VAR_*'], 'w'))

@2 <summary <difficulty >recommendation

python <recommendation  (print)

```

Code:

```python >input

print("PyTorch is an open-source machine learning framework developed by Meta AI Research. It provides dynamic computational graphs, automatic differentiation, and seamless GPU acceleration. PyTorch is widely used in research for flexibility and in production for TorchScript and TorchServe capabilities.")

```

<!--@1 <input >raw_response

Respond in exactly this format (3 lines):

SUMMARY: one sentence paraphrase

TAGS: tag1, tag2, tag3, tag4

DIFFICULTY: one word \u2014 beginner / intermediate / expert

-->

```python <raw_response >summary >tags >difficulty
import os, re
text = open(os.environ['MSH_VAR_raw_response']).read()
summary = re.search(r'SUMMARY:\s*(.+)', text)
tags = re.search(r'TAGS:\s*(.+)', text)
difficulty = re.search(r'DIFFICULTY:\s*(.+)', text)
summary = summary.group(1).strip() if summary else "n/a"
tags = tags.group(1).strip() if tags else "n/a"
difficulty = difficulty.group(1).strip() if difficulty else "n/a"
with open(os.environ['MSH_VAR_summary'], 'w') as f: f.write(summary)
with open(os.environ['MSH_VAR_tags'], 'w') as f: f.write(tags)
with open(os.environ['MSH_VAR_difficulty'], 'w') as f: f.write(difficulty)
print(f"Summary   : {summary}")
print(f"Tags      : {tags}")
print(f"Difficulty: {difficulty}")
```

<!--@2 <summary <difficulty >recommendation

The first input is a library summary. The second is its difficulty level. Give a one-sentence personalized learning path recommendation.

-->

```python <recommendation
import os
print("\n=== Learning Recommendation ===")
print(open(os.environ['MSH_VAR_recommendation']).read().strip())
```

PyTorch is an open-source machine learning framework developed by Meta AI Research. It provides dynamic computational graphs, automatic differentiation, and seamless GPU acceleration. PyTorch is widely used in research for flexibility and in production for TorchScript and TorchServe capabilities.

SUMMARY: PyTorch is Meta's open-source machine learning framework offering dynamic graphs, automatic differentiation, and GPU support for both research and production use.

TAGS: pytorch, machine-learning, deep-learning, meta

DIFFICULTY: intermediate

Summary   : PyTorch is Meta's open-source machine learning framework offering dynamic graphs, automatic differentiation, and GPU support for both research and production use.

Tags   : pytorch, machine-learning, deep-learning, meta

Difficulty: intermediate

Since you\u2019re at an intermediate level, start by re\u2011implementing a few classic models (e.g., MLPs and CNNs) in PyTorch to get comfortable with tensors, autograd, and the training loop, then move on to writing custom `nn.Module`s, experimenting with dynamic control flow, and finally deploying a small GPU\u2011accelerated project (like an image classifier or text classifier) to solidify both research-style prototyping and basic production use.

/home/igor >

=== Learning Recommendation ===

Since you’re at an intermediate level, start by reimplementing a few classic models (e.g., MLPs and CNNs) in PyTorch to get comfortable with tensors, autograd, and the training loop, then move on to writing custom `nn.Module`s, experimenting with dynamic control flow, and finally deploying a small GPU\u2011accelerated project (like an image classifier or text classifier) to solidify both research-style prototyping and basic production use.

/home/igor >

/home/igor > Sent to mshell (2994 bytes)

Received from GUI editor:

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

What it does: CONFIG documents pipeline parameters. WHILE loop runs until explanation quality >= threshold. LLM @1 generates a 3-sentence explanation with code. LLM @2 scores it 1\u201310. Python checks threshold. After loop LLM @3 polishes for publication. Python prints final result with metrics.

Key patterns:

- @2 receives `<target_audience` explicitly \u2014 scorer needs context.

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

- CONFIG makes the pipeline reusable \u2014 change two lines to switch topic and audience.

Flow diagram:

```

config  (algorithm, target_audience, quality_threshold)

python >algorithm, >target_audience, >status, >iteration, >quality, >explanation

[WHILE status:running]

  python <iteration >iteration

  @1 <algorithm <target_audience >explanation      (generate)

  @2 <explanation <target_audience >quality       (score 1-10)

  python <iteration <quality >status             (log + check)

[END_WHILE]

@3 <explanation >final_polish                    (polish)

python <final_polish <quality <iteration          (print)

```

Code:

```config

algorithm=binary search

target_audience=first-year computer science student

quality_threshold=7

```

```python >algorithm

print("binary search")

```

```python >target_audience

print("first-year computer science student")

```

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

```python >iteration
print("0")
```

```python >quality
print("0")
```

```python >explanation
print("")
```

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

```python <iteration >iteration
import os
val = int(open(os.environ['MSH_VAR_iteration']).read().strip())
print(val + 1)
```

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

The first input is an algorithm. The second is the target audience. Explain it in exactly 3 sentences with no jargon and a tiny Python code snippet (2-3 lines).

-->

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

Rate this explanation for the target audience 1-10. Criteria: clarity, accuracy, usefulness of code example. Reply with ONLY the integer score.

-->

```python <iteration <quality >status
import os
itr = open(os.environ['MSH_VAR_iteration']).read().strip()
```

```
q = open(os.environ['MSH_VAR_quality']).read().strip()
print(f"[Iter {itr}] Quality: {q}")
try:
    print("done" if int(q) >= 7 else "running")
except ValueError:
    print("running")
```

<!--@end_while-->

<!--@3 <explanation >final_polish

Polish this explanation for publication in a Python learning course. Keep 3 sentences plus the code snippet. Professional tone, no markdown.

-->

```python <final_polish <quality <iteration
import os
q = open(os.environ['MSH_VAR_quality']).read().strip()
itr = open(os.environ['MSH_VAR_iteration']).read().strip()
print(f"=== Final (score={q}, iters={itr}) ===")
print(open(os.environ['MSH_VAR_final_polish']).read().strip())
```

[config] algorithm=binary search

[config] target_audience=first-year computer science student

[config] quality_threshold=7

binary search

first-year computer science student

running

0

0

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

1

Binary search is a way to find a specific item in a sorted list by repeatedly checking the middle element and eliminating half of the remaining possibilities each time. If the middle element is too big, you search the left half; if it's too small, you search the right half. This process continues until you find your target or determine it's not in the list.

```python
mid = (left + right) // 2

if arr[mid] == target: return mid
```

9

[Iter 1] Quality: 9

done

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

Binary search efficiently locates a target value in a sorted list by repeatedly dividing the search interval in half based on comparisons with the middle element. The algorithm adjusts the interval boundaries to the left or right subinterval where the target must reside if present, eliminating half the remaining elements each iteration. The process terminates when the target is found at the midpoint or the interval is empty, confirming the target's absence.

mid = (left + right) // 2

if arr[mid] == target: return mid

/home/igor > === Final (score=9, iters=1) ===

Binary search efficiently locates a target value in a sorted list by repeatedly dividing the search interval in half based on comparisons with the middle element. The algorithm adjusts the interval boundaries to the left or right subinterval where the target must reside if present, eliminating half the remaining elements each iteration. The process terminates when the target is found at the midpoint or the interval is empty, confirming the target's absence.

mid = (left + right) // 2

if arr[mid] == target: return mid

/home/igor >

/home/igor > Sent to mshell (2196 bytes)

Received from GUI editor:

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

What it does: Python creates a list of 5 Python module names \u2014 some valid, some broken. FOREACH iterates. Per item TRY attempts `importlib.import_module`. If success, Python stores module info and LLM describes its use. If import fails, CATCH prints `[ERR]` as a hardcoded literal. Loop continues regardless of errors.

Key patterns:

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

- CATCH block does not use `<module_error` \u2014 prints the literal string directly.

- Do not create dependencies on variables written inside TRY \u2014 if TRY fails they are not written.

Flow diagram:

```

python >modules = "os\ncollections\nfakemodule_xyz\njson\nnonexistent_pkg"

[FOREACH module in modules]

  [TRY]

    python <module >module_info       (importlib.import_module \u2014 may fail)

    @1 <module_info >insight         (LLM: describe the module)

    python <insight                 (print [OK])

  [CATCH >module_error]

    python                          (print [ERR] literal string)

  [END_TRY]

[END_FOREACH]

python  (completion message)

```

Code:

```python >modules

print("os\ncollections\nfakemodule_xyz\njson\nnonexistent_pkg")
```

````
```

<!--@foreach module in modules-->

<!--@try-->

```python <module >module_info

import os, importlib

module_name = open(os.environ['MSH_VAR_module']).read().strip()

mod = importlib.import_module(module_name)

doc_line = (mod.__doc__ or 'No docstring').strip().split('\n')[0][:80]

attrs = len([a for a in dir(mod) if not a.startswith('_')])

print(f"Module: {module_name} | Public attrs: {attrs} | Doc: {doc_line}")

```

<!--@1 <module_info >insight

The input contains metadata about a Python standard library module. In one sentence,
describe its primary real-world use case.

-->

```python <insight

import os

print(f"[OK] {open(os.environ['MSH_VAR_insight']).read().strip()}")

```

<!--@catch >module_error-->

```python

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

```

<!--@end_try-->

<!--@end_foreach-->

```python

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

```
````
````

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

os

collections

fakemodule_xyz

json

nonexistent_pkg

[foreach] iter 1: module=os

[try] executing try block

Module: os | Public attrs: 384 | Doc: OS routines for NT or Posix depending on what system we're on.

The os module is primarily used for interacting with the operating system to perform file and directory operations like creating folders, reading file paths, and running system commands.

[OK] The os module is primarily used for interacting with the operating system to perform file and directory operations like creating folders, reading file paths, and running system commands.

[try] try block succeeded

[foreach] iter 2: module=collections

[try] executing try block

Module: collections | Public attrs: 9 | Doc: This module implements specialized container datatypes providing

The collections module is primarily used to provide specialized data structures like Counter for counting items, defaultdict for dictionaries with default values, and deque for efficient queue operations that aren't available with basic Python types.

[OK] The collections module is primarily used to provide specialized data structures like Counter for counting items, defaultdict for dictionaries with default values, and deque for efficient queue operations that aren't available with basic Python types.

[try] try block succeeded

[foreach] iter 3: module=fakemodule_xyz

[try] executing try block

Traceback (most recent call last):

  File "/tmp/mshell_script_315148.py", line 3, in <module>

mod = importlib.import_module(module_name)

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

File "/usr/lib/python3.12/importlib/__init__.py", line 90, in import_module

return _bootstrap._gcd_import(name[level:], package, level)

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

File "<frozen importlib._bootstrap>", line 1387, in _gcd_import

File "<frozen importlib._bootstrap>", line 1360, in _find_and_load

File "<frozen importlib._bootstrap>", line 1324, in _find_and_load_unlocked

ModuleNotFoundError: No module named 'fakemodule_xyz'

The collections module is primarily used to provide specialized container data types like Counter, defaultdict, and deque that offer more functionality and better performance than basic Python lists and dictionaries for specific use cases.

[try] try block failed, executing catch block

[ERR] Import failed: try_block_failed

[foreach] iter 4: module=json

[try] executing try block

Module: json | Public attrs: 12 | Doc: JSON (JavaScript Object Notation) <https://json.org> is a subset of

The json module is primarily used for converting Python data structures to JSON strings and vice versa, enabling data exchange between Python applications and web APIs or configuration files.

[OK] The json module is primarily used for converting Python data structures to JSON strings and vice versa, enabling data exchange between Python applications and web APIs or configuration files.

[try] try block succeeded

[foreach] iter 5: module=nonexistent_pkg

[try] executing try block

Traceback (most recent call last):

File "/tmp/mshell_script_315148.py", line 3, in <module>

mod = importlib.import_module(module_name)

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

File "/usr/lib/python3.12/importlib/__init__.py", line 90, in import_module

return _bootstrap._gcd_import(name[level:], package, level)

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

File "<frozen importlib._bootstrap>", line 1387, in _gcd_import

File "<frozen importlib._bootstrap>", line 1360, in _find_and_load

File "<frozen importlib._bootstrap>", line 1324, in _find_and_load_unlocked

ModuleNotFoundError: No module named 'nonexistent_pkg'

The json module is primarily used for parsing JSON data from APIs and files into Python objects, and converting Python data back to JSON format for storage or transmission.

[try] try block failed, executing catch block

[ERR] Import failed: try_block_failed

/home/igor > === Batch complete. Errors were isolated, pipeline never stopped. ===

/home/igor >

---

References:

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

---