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

Pure C++ language Edition — Art2Dec SoftLab, March 17th 2026

Pure Edition C++ -Only · No Python, C,,Rust, Go, Lua or Bash — only C++, `mshell` directives, and LLM blocks. Based on the original mshell Workflow Patterns Reference Guide Parts I & II, C++ language use idiomatic C++17.

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 uses C++ exclusively for all code blocks.

Quick Reference

Reading a Variable in C++

```
// slurp entire file to string
std::ifstream f(std::getenv("MSH_VAR_varname"));
std::string content((std::istreambuf_iterator<char>(f)), {});
```

Writing a Variable Directly (required for multiple >outvar)

```
std::ofstream w(std::getenv("MSH_VAR_varname"));
w << "value\n";
w.close();
```

Reusable slurp() Helper

```
std::string slurp(const char* key) {
    std::ifstream f(std::getenv(key));
    return {std::istreambuf_iterator<char>(f), {}};
}
```

Compiling & Running LLM-Generated Code

```
const char* path = std::getenv("MSH_VAR_code");
// copy to stable .cpp path — g++ needs proper extension to recognise C++ sou
rce
```

```cpp
std::string cp_cmd = std::string("cp -- \"") + path + "\" /tmp/msh_gen.cpp";
std::system(cp_cmd.c_str());
if (std::system("g++ -std=c++17 -O2 -o /tmp/msh_bin /tmp/msh_gen.cpp 2>&1") !
= 0) {
    std::cerr << "Compile failed\n"; return 1;
}
std::system("/tmp/msh_bin");
```

Stripping Whitespace (equivalent of `tr -d '[:space:]'`)

```cpp
str.erase(std::remove_if(str.begin(), str.end(),
          [](char c){ return std::isspace((unsigned char)c); }),
          str.end());
```

Node Syntax Reference

Node	Syntax	Semantics
WHILE	`<!--@while var:value--> … <!--@end_while-->`	Loop while var == value
FOREACH	`<!--@foreach item in listvar--> … <!--@end_foreach-->`	Iterate over lines of variable
TRY/CATCH	`<!--@try--> … <!--@catch >errvar--> … <!--@end_try-->`	Error isolation
SPLIT	`<!--@split var into N-->`	Visual marker; creates var_1, var_2, …
MERGE	`<!--@merge-->`	Visual merge marker
CONFIG	```` ```config … ``` ````	Parameter documentation block
Async LLM	`<!--@1 <in >out async-->`	Run LLM in background process
Await	```` ```cpp await=var1,var2 ````	Synchronisation barrier

FOREACH: list must be produced with one item per line (`\n`-separated).
WHILE: condition checked before body; body must modify the condition variable.
CONFIG: does not inject variables at runtime — always pair with `cpp` blocks.
CATCH >errvar: variable receives the literal string `"try_block_failed"`. Do NOT use `<errvar` inside the CATCH block — print `"try_block_failed"` directly.
Multiple `>outvar` on C++ block: block must write to `$MSH_VAR_*` files directly.
Async job matching: each async call must have exactly one `>outvar`.

C++ Idiom Cheatsheet for mshell

Task	C++ idiom
Read variable	`std::ifstream f(std::getenv("MSH_VAR_x")); std::string`

Task	C++ idiom
	`s((std::istreambuf_iterator<char>(f)), {});`
Write variable	`std::ofstream w(std::getenv("MSH_VAR_x")); w << val;`
Compile LLM code	`cp MSH_VAR_code to .cpp, then: std::system("g++ -std=c++17 -O2 -o /tmp/bin /tmp/gen.cpp 2>&1")`
Run compiled binary	`std::system("/tmp/bin")`
Strip whitespace	`str.erase(std::remove_if(…std::isspace…), str.end())`
Parse structured output	`std::regex re("LABEL:\\s*(.+)"); std::smatch m; std::regex_search(text, m, re);`
Exit to trigger CATCH	`std::exit(1)` or `return 1;` from `main()`
Non-zero exit = CATCH	Any exit code ≠ 0 jumps to `<!--@catch-->`

Pattern Summary Table

#	Pattern	Node Types	Models	Key Feature
1	Linear Pipeline	—	—	Sequential C++ stages
2	LLM in Middle	—	@1	LLM as transformer between C++ stages
3	Fan-Out	—	@1	One source, many C++ consumers
4	Code Gen + Exec	—	@1	LLM generates C++ → g++ → run
5	Two-LLM Review	—	@1, @2	Generate → Review → Improve → Compile
6	Parallel 3 Models	—	@1,@2,@3	Same query to 3 models
7	Eval-Optimizer Loop	LOOP	@1, @2	Loop until ACCEPTED
8	Multi-Stage + Multi-Model	—	@1, @2	Full C++ pipeline + LLM analysis
9	Routing	—	@1	LLM classifies → conditional C++ branch
10	Full Pipeline	SPLIT, MERGE, AWAIT	@1,@2,@3	All patterns combined
11	MShell Node	mshell	@1, @2	Native mshell AI calls + C++ I/O
12	Async + Await +	AWAIT	@1,@2,@3	Parallel async + barrier

#	Pattern	Node Types	Models	Key Feature
	Synthesis			
13	WHILE Counter	WHILE	@1	Flag-based exit + LLM inside loop
14	FOREACH List	FOREACH	@1	Line-by-line list processing
15	TRY/CATCH	TRY/CATCH	—	exit(1); CATCH without <errvar
16	SPLIT + MERGE	SPLIT, MERGE	@1, @2	Async parallel analysis + synthesis
17	CONFIG Pipeline	CONFIG	@1, @2	Parameterised reusable template
18	FOREACH + Async	FOREACH, AWAIT	@1, @2	Per-item parallel LLM processing
19	WHILE Quality Gate	WHILE	@1, @2	Numeric threshold exit condition
20	Map-Reduce	SPLIT, MERGE, AWAIT	@1, @2	Async map + reduce synthesis
21	TRY/CATCH + LOOP	TRY/CATCH, LOOP	@1	Retry with self-correction
22	Multi-Var Output	—	@1, @2	Structured field extraction via C++ regex
23	CONFIG+WHILE+3M	CONFIG, WHILE	@1–@3	Fully adaptive pipeline with scorer
24	FOREACH+TRY/CATCH	FOREACH, TRY/CATCH	@1	Fault-tolerant batch; CATCH without <errvar

---

Part I — Patterns 1–12: Core Patterns

Pattern 1 — Linear Data Pipeline (Pure C++)

Concept: Sequential data flow where each C++ block reads the previous block's output and passes its own result forward. Demonstrates that a complete, multi-stage computation can be expressed in a single mshell document using only one language.

Use case: Multi-stage numerical pipelines — raw generation → statistics → formatting → analysis → aggregation → reporting, each stage a standalone C++ binary.

Key concept: Variables flow strictly top-to-bottom. Each `>outvar` producer must appear before any `<outvar` consumer in the document.

Flow Diagram

```
cpp >raw_number
  ↓
cpp <raw_number >squared_stats      (compute square, cube, sqrt)
  ↓
cpp <squared_stats >formatted       (format as a report string)
  ↓
cpp <formatted >enriched            (append prime-check result)
  ↓
cpp <enriched >filtered             (uppercase all letters)
  ↓
cpp <filtered >final_report         (wrap in decorative frame)
  ↓
cpp <final_report                   (print to stdout)
```

Code

```
cpp >raw_number #include <iostream> int main() {     std::cout << 42 <<
std::endl;     return 0; }
```

```
cpp <raw_number >squared_stats #include <iostream> #include <fstream>
#include <cmath> #include <string> int main() {     std::ifstream
f(std::getenv("MSH_VAR_raw_number"));     int val; f >> val;     long long sq
= (long long)val * val;     long long cu = sq * val;     double sr =
std::sqrt((double)val);     std::cout << "value=" << val               << "
square=" << sq               << " cube=" << cu               << " sqrt=" <<
sr               << std::endl;     return 0; }
```

```
cpp <squared_stats >formatted #include <iostream> #include <fstream> #include
<sstream> #include <string> int main() {     std::ifstream
f(std::getenv("MSH_VAR_squared_stats"));     std::string line;
std::getline(f, line);     std::cout << "[Stage 3 / Formatted] " << line <<
std::endl;     return 0; }
```

```
cpp <formatted >enriched #include <iostream> #include <fstream> #include
<string> #include <sstream> // naive primality check extracted from the stats
value bool is_prime(int n) {     if (n < 2) return false;     for (int i = 2;
i * i <= n; i++) if (n % i == 0) return false;     return true; } int main()
{     std::ifstream f(std::getenv("MSH_VAR_formatted"));     std::string
line; std::getline(f, line);     // extract the raw value from "value=N ..."
int val = 0;     std::istringstream ss(line);     std::string tok;     while
(ss >> tok) {         if (tok.rfind("value=", 0) == 0) val =
std::stoi(tok.substr(6));     }     std::string prime_label = is_prime(val) ?
```

```
"prime" : "composite";     std::cout << line << " prime=" << prime_label <<
std::endl;     return 0; }
```

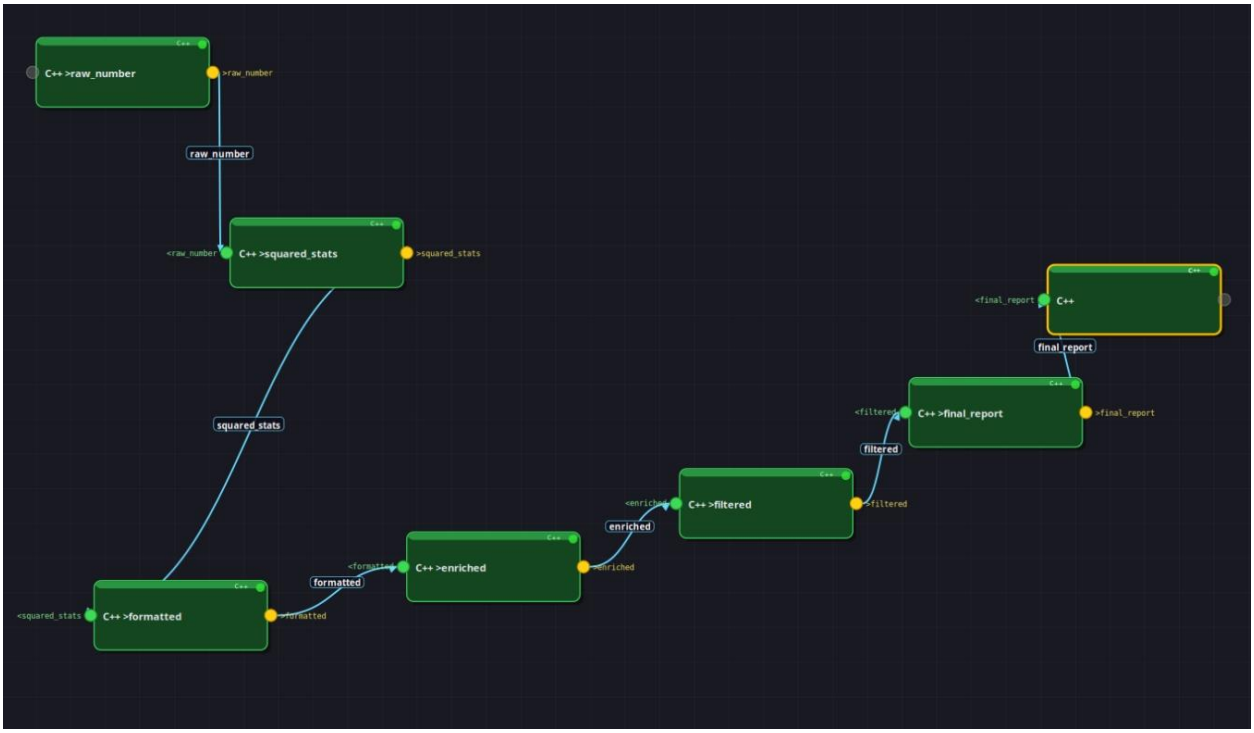
```
cpp <enriched >filtered #include <iostream> #include <fstream> #include
<string> #include <cctype> int main() {     std::ifstream
f(std::getenv("MSH_VAR_enriched"));     std::string line; std::getline(f,
line);     for (char& c : line) c = std::toupper(c);     std::cout << line <<
std::endl;     return 0; }
```

```
cpp <filtered >final_report #include <iostream> #include <fstream> #include
<string> int main() {     std::ifstream f(std::getenv("MSH_VAR_filtered"));
std::string line; std::getline(f, line);     std::string border(line.size() +
4, '=');     std::cout << border << "\n"              << "| " << line << "
|\n"              << border << std::endl;     return 0; }
```

```
cpp <final_report #include <iostream> #include <fstream> #include <string>
int main() {     std::ifstream f(std::getenv("MSH_VAR_final_report"));
std::string line;     while (std::getline(f, line)) std::cout << line <<
"\n";     return 0; }
```

Key Patterns

- Every stage is a self-contained g++-compiled binary; mshell compiles and runs each block automatically.
- `std::getenv("MSH_VAR_varname")` returns the **file path** of the variable — open it with `std::ifstream`.
- No shared memory, no global state — pure file-based data flow.



Pattern 2 — LLM in the Middle

Concept: A C++ block generates structured data, an LLM transforms it with natural language intelligence, and a second C++ block processes the model's response for further use.

Use case: Automated log analysis — C++ generates a synthetic server-error log, the LLM produces a human-readable incident summary, and C++ extracts word/sentence statistics from the summary.

Key concept: LLM directive `<!--@1 <data >result ... -->` injects the variable file contents into the prompt and writes the model response to `>result`, which any subsequent block can consume exactly like any other variable.

Flow Diagram

```
cpp >log_data           (generate synthetic error log)
  ↓
@1 <log_data >summary   (LLM: summarise the incident in plain English)
  ↓
cpp <summary            (count words & sentences, print report)
```

Code

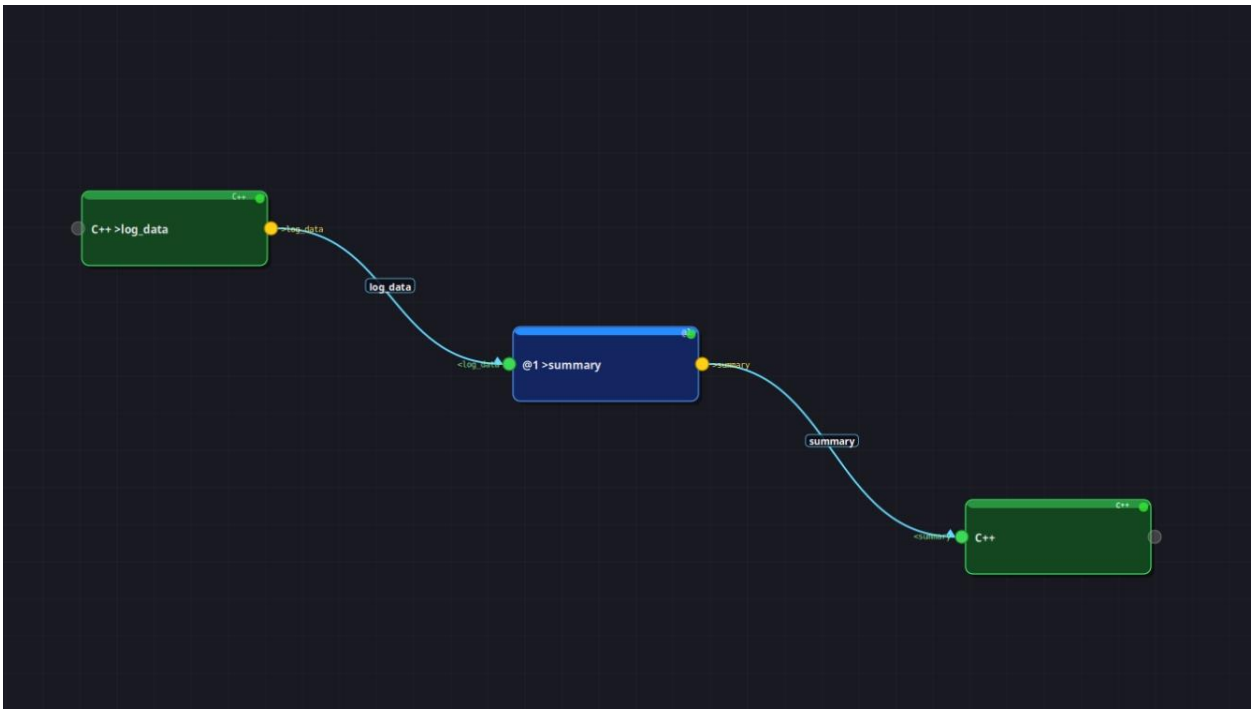
```
cpp >log_data #include <iostream> int main() {     std::cout <<
"2026-03-11 02:14:33 ERROR  [auth-service]   Connection timeout after 30s
(attempt 3/3)\n"         "2026-03-11 02:14:35 ERROR  [db-pool]        Max
connections reached (limit=100)\n"         "2026-03-11 02:14:36 WARN   [api-
gateway]    Retrying upstream /users endpoint\n"         "2026-03-11 02:14:40
ERROR  [auth-service]   JWT validation failed: token expired\n"
"2026-03-11 02:14:41 FATAL  [scheduler]      Worker thread panic:
segmentation fault\n"         "2026-03-11 02:14:42 INFO   [watchdog]
Restarting auth-service (pid 4872)\n";     return 0; }
```

```
cpp <summary #include <iostream> #include <fstream> #include <sstream>
#include <string> int main() {     std::ifstream
f(std::getenv("MSH_VAR_summary"));     std::string
text((std::istreambuf_iterator<char>(f)),
std::istreambuf_iterator<char>());     // count words     std::istringstream
ss(text);     std::string tok;     int words = 0;     while (ss >> tok)
++words;     // count sentences (. ! ?)     int sentences = 0;     for (char
c : text) if (c == '.' || c == '!' || c == '?') ++sentences;     std::cout <<
"=== LLM Incident Summary ===\n" << text << "\n"               << "--- Meta:
```

```
" << words << " words, ~"              << sentences << " sentence(s) ---\n";
return 0; }
```

Key Patterns

- The LLM receives the **entire file content** of `log_data` as part of the prompt.
- C++ reads the LLM response file via `std::getenv("MSH_VAR_summary")` — same mechanism as any other variable.
- `std::istreambuf_iterator` is the idiomatic way to slurp a whole file into a `std::string` in C++.



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

Concept: A single variable is produced once and independently consumed by multiple C++ blocks and an LLM — each performing a different analysis without modifying the shared source.

Use case: Multi-perspective analysis of a text corpus — one C++ block counts character-level stats, another computes word frequency, a third finds the longest word, and an LLM judges the writing style, all from the same input string.

Key concept: Multiple blocks can reference the same `<varname`. mshell never modifies a variable file on read — all consumers see the original, unchanged content.

Flow Diagram

```
cpp >text                          (produce the source text)
  ↓          ↓           ↓         ↓
cpp <text   cpp <text   cpp <text   @1 <text
(char stats)(word freq) (longest)   (style judge)
```

Code

```
cpp >text #include <iostream> int main() {     std::cout <<         "The
greatest glory in living lies not in never falling, "         "but in rising
every time we fall. "         "The way to get started is to quit talking and
begin doing. "         "Life is what happens when you are busy making other
plans."         << std::endl;     return 0; }
```

```
cpp <text // Consumer 1: character-level statistics #include <iostream>
#include <fstream> #include <string> #include <cctype> int main() {
std::ifstream f(std::getenv("MSH_VAR_text"));     std::string
text((std::istreambuf_iterator<char>(f)),
std::istreambuf_iterator<char>());     int letters = 0, digits = 0, spaces =
0, punct = 0;     for (char c : text) {         if (std::isalpha(c))
++letters;         else if (std::isdigit(c)) ++digits;         else if
(std::isspace(c)) ++spaces;         else if (std::ispunct(c)) ++punct;     }
std::cout << "[C++ Char Stats] letters=" << letters               << "
spaces=" << spaces               << " punct=" << punct               << "
total=" << text.size() << "\n";     return 0; }
```

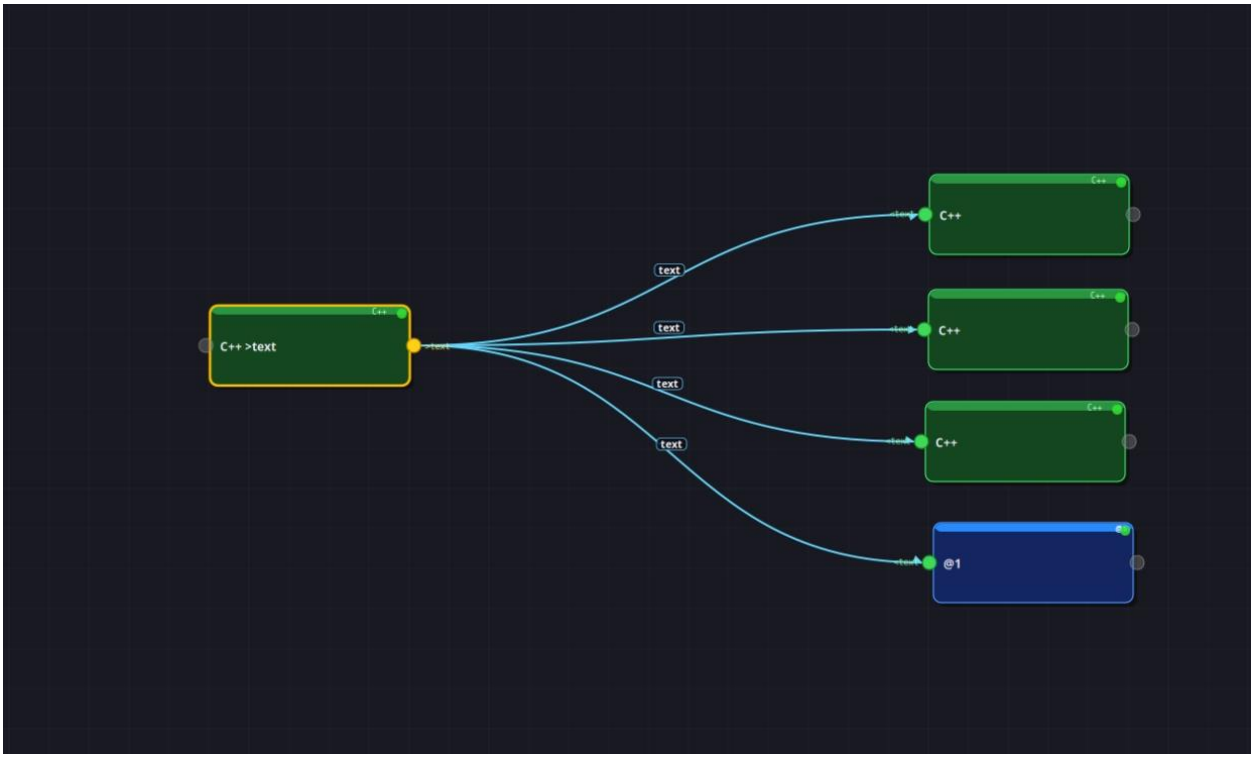
```
cpp <text // Consumer 2: top-3 most frequent words #include <iostream>
#include <fstream> #include <sstream> #include <string> #include <map>
#include <vector> #include <algorithm> #include <cctype> int main() {
std::ifstream f(std::getenv("MSH_VAR_text"));     std::string
text((std::istreambuf_iterator<char>(f)),
std::istreambuf_iterator<char>());     // normalise: strip punctuation,
lowercase     std::string clean;     for (char c : text) clean +=
std::isalpha(c) ? std::tolower(c) : ' ';     std::istringstream ss(clean);
std::map<std::string, int> freq;     std::string w;     while (ss >> w)
++freq[w];     std::vector<std::pair<int,std::string>> ranked;     for (auto&
p : freq) ranked.push_back({p.second, p.first});
std::sort(ranked.rbegin(), ranked.rend());     std::cout << "[C++ Word Freq]
top words: ";     for (int i = 0; i < std::min(3,(int)ranked.size()); ++i)
std::cout << ranked[i].second << "(" << ranked[i].first << ") ";
std::cout << "\n";     return 0; }
```

```
cpp <text // Consumer 3: longest word #include <iostream> #include <fstream>
#include <sstream> #include <string> #include <cctype> int main() {
std::ifstream f(std::getenv("MSH_VAR_text"));     std::string
text((std::istreambuf_iterator<char>(f)),
std::istreambuf_iterator<char>());     std::string clean;     for (char c :
```

```
text) clean += std::isalpha(c) ? std::tolower(c) : ' ';
std::istringstream ss(clean);     std::string w, longest;     while (ss >> w)
if (w.size() > longest.size()) longest = w;     std::cout << "[C++ Longest]
\"" << longest               << "\" (" << longest.size() << " chars)\n";
return 0; }
```

Key Patterns

- Four independent blocks all read `MSH_VAR_text` — none of them writes back to it.
- Execution is sequential in document order, but the source file is never mutated.
- Each C++ binary is compiled independently; they share no state beyond the variable file.



Pattern 4 — LLM Code Generation → Execute via Variable

Concept: An LLM generates executable C++ source code from a natural-language task description. A C++ "launcher" block reads the generated file path from `MSH_VAR_code`, compiles it with g++, and runs the resulting binary — all without any manual copy-paste.

Use case: On-the-fly algorithm generation — ask the model to implement a sorting algorithm, instantly compile and benchmark it.

Key concept: `MSH_VAR_code` is a **file path**. Compile with `cp MSH_VAR_code /tmp/gen.cpp && g++ -O2 -o /tmp/gen_bin /tmp/gen.cpp && /tmp/gen_bin` to turn LLM output into a running program.

Flow Diagram

```
cpp >task               (describe the coding task)
  ↓
@1 <task >code          (LLM: write C++ source, no fences)
  ↓
cpp <code               (compile with g++, run, print output)
```

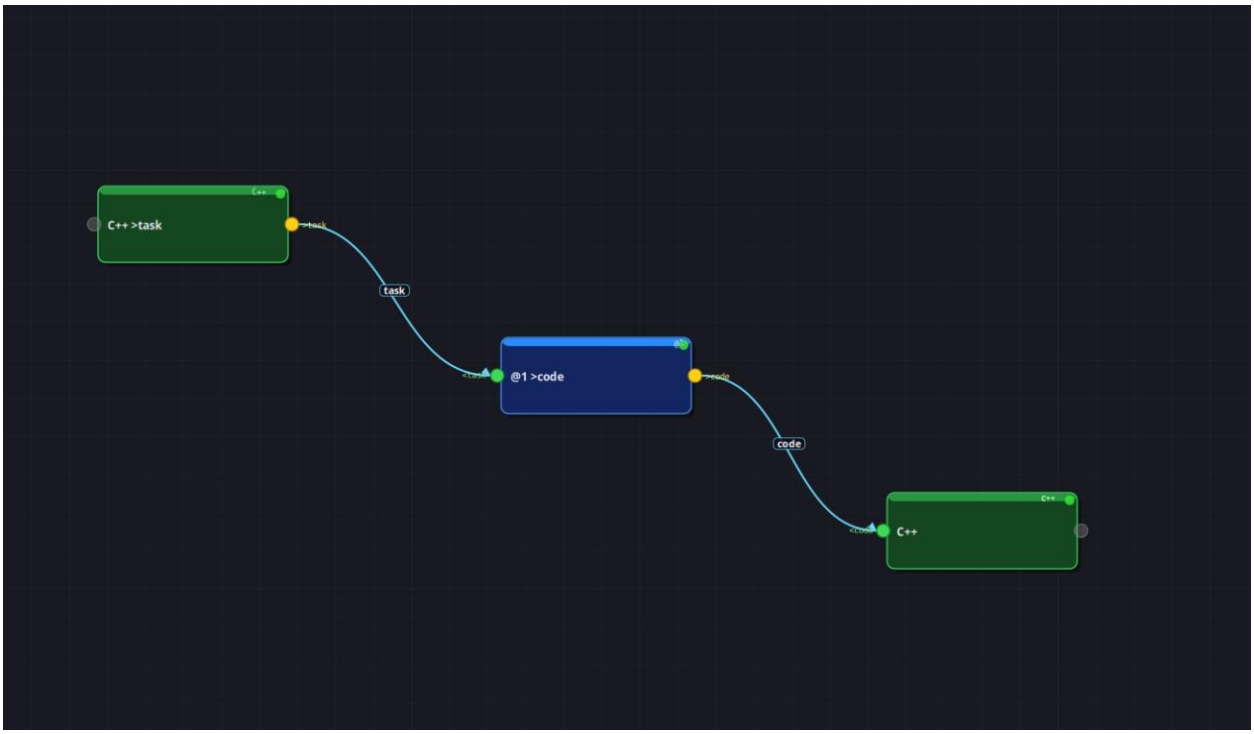
Code

```
cpp >task #include <iostream> int main() {     std::cout <<         "Write a C++ program that generates the first 15 Fibonacci numbers, "         "prints them one per line, and then prints their sum. "         "Use only the standard library. No classes needed."         << std::endl;     return 0; }
```

```cpp <code #include  #include  #include  #include  int main() { const char* code_path = std::getenv("MSH_VAR_code");

// copy to stable .cpp path — g++ needs a proper extension to recognise C++ source
std::string cp_cmd = std::string("cp -- \"") + code_path + "\" /tmp/msh_p4_gen.cpp";
std::system(cp_cmd.c_str());

std::cout << "=== Generated C++ source ===\n";
std::ifstream src("/tmp/msh_p4_gen.cpp");
std::string line;
while (std::getline(src, line)) std::cout << line << "\n";
src.close();

std::cout << "\n=== Compiling & Running ===\n";
int ret = std::system(
    "g++ -std=c++17 -O2 -o /tmp/msh_p4_bin /tmp/msh_p4_gen.cpp 2>&1");
if (ret != 0) {
    std::cerr << "Compilation failed\n";
    return 1;
}
std::system("/tmp/msh_p4_bin");
return 0;

}


---
```

Key Patterns

- `MSH_VAR_code` holds the **path** to a temp file containing the LLM's text output.
- `std::system()` lets C++ drive the shell — compile then execute in one block.
- Always add `2>&1` to the compile command so errors are visible in the pipeline log.
- Use `-std=c++17 -O2` as the default compile flags; adjust per task.



Pattern 5 — Two-LLM Review Chain

Concept: Model @1 generates an initial C++ solution. Model @2 reviews it for
correctness, safety, and efficiency. Model @1 then receives both the original code and the
review and produces an improved version. A final C++ block compiles and runs the result.

Use case: Automated C++ code quality improvement — generate a memory-safe implementation, have a second model critique it (looking for undefined behaviour, missing

bounds checks, etc.), then refine and execute.

Key concept: The refinement LLM call passes both `<code` and `<review` as inputs.
When multiple `<invar` are present mshell labels each section with `[varname]:` in the
prompt automatically.

Flow Diagram

cpp >task ↓ @1

code (generate initial C++ implementation) ↓ cpp <code (print generated code) ↓ @2 review (critique: correctness, UB, efficiency — 2 sentences) ↓ cpp <review (print review) ↓ @1 <code improved (refine based on review) ↓ cpp <improved (compile with g++, execute, print result)

Code

```cpp >task
#include <iostream>
int main() {
    std::cout <<
        "Write a C++ function bool is_palindrome(const std::string& s) "
        "that returns true if s reads the same forwards and backwards, "
        "ignoring case and non-alphabetic characters. "
        "Include a main() that tests it on at least 4 examples."
        << std::endl;
    return 0;
}
```

cpp <code #include <iostream> #include <fstream> #include <string> int main() { std::ifstream f(std::getenv("MSH_VAR_code")); std::string line; std::cout << "=== Model @1 Generated ===\n"; while (std::getline(f, line)) std::cout << line << "\n"; return 0; }

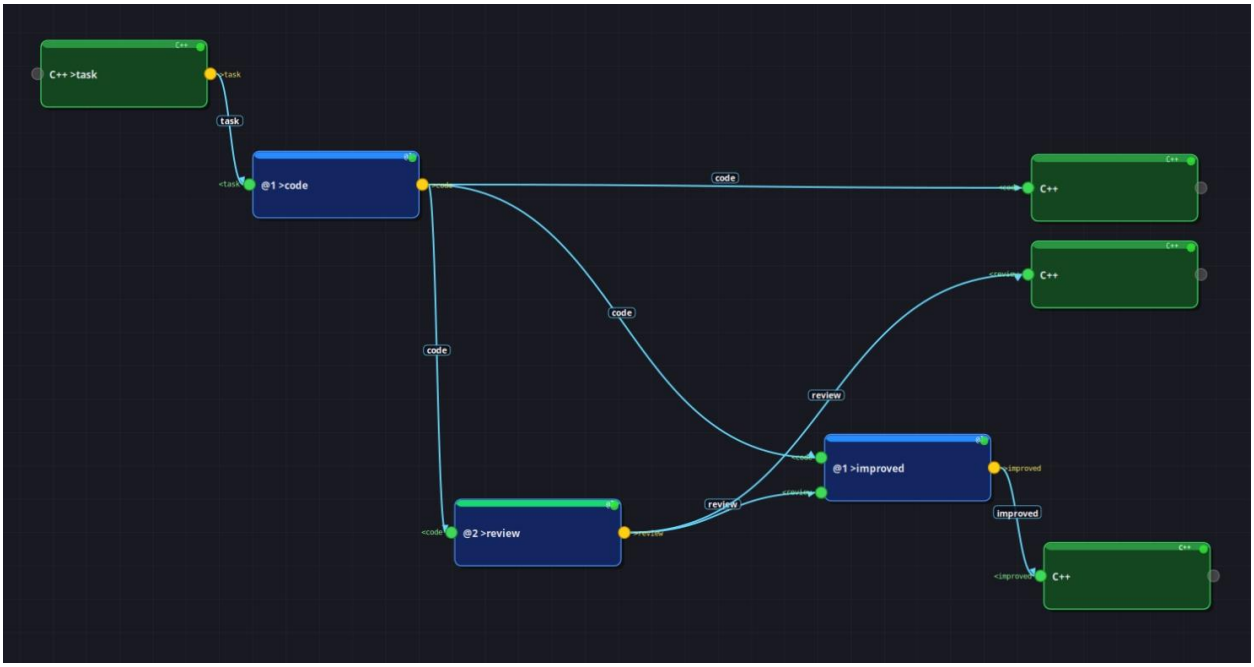
cpp <review #include <iostream> #include <fstream> #include <string> int main() { std::ifstream f(std::getenv("MSH_VAR_review")); std::string line; std::cout << "=== Model @2 Review ===\n"; while (std::getline(f, line)) std::cout << line << "\n"; return 0; }

```cpp <improved #include #include #include #include int main() { const char* path = std::getenv("MSH_VAR_improved"); std::cout << "=== Final Improved Code ==="; std::ifstream src(path); std::string line; while (std::getline(src, line)) std::cout << line << ""; src.close();
```

```
std::cout << "\n=== Compiling & Executing ===\n";
// copy to stable .cpp path before compiling
std::string cp_cmd = std::string("cp -- \"") + path + "\" /tmp/msh_p5_gen.cpp
";
std::system(cp_cmd.c_str());
if (std::system("g++ -std=c++17 -O2 -o /tmp/msh_p5_bin /tmp/msh_p5_gen.cpp 2>
&1") != 0) {
    std::cerr << "Compile failed\n"; return 1;
}
std::system("/tmp/msh_p5_bin");
return 0;

}
```

Key Patterns

- **Role separation:** @1 = generator/fixer, @2 = critic. Neither model sees the other's
  system instructions.
- Multiple `<invar` on an LLM directive injects each variable with a `[varname]:` label,
  letting the model distinguish the original code from the review.
- The final C++ block uses `std::system()` to drive compilation — same pattern as
  Pattern 4.



Pattern 6 — Parallel 3-Model Query

Concept: The same C++-generated question is sent to all three LLM models independently. Each model stores its answer in a separate variable. A final C++ block
collects and formats all three responses side-by-side.

Use case: Ensemble design advice — ask three models to recommend the best C++ data
structure for a given problem, then compare their reasoning.

Key concept: Three separate LLM directive blocks with `@1`, `@2`, `@3` all read the
same `<question` and write to distinct `>ans1`, `>ans2`, `>ans3` output variables.
Execution is sequential in document order but each model is independent.

Flow Diagram

cpp >question ↓ ↓ ↓ @1 <question @2 <question @3 ans1 >ans2 >ans3 ↓ ↓ ↓ cpp <ans1 <ans2 <ans3 (print all three side-by-side)

Code

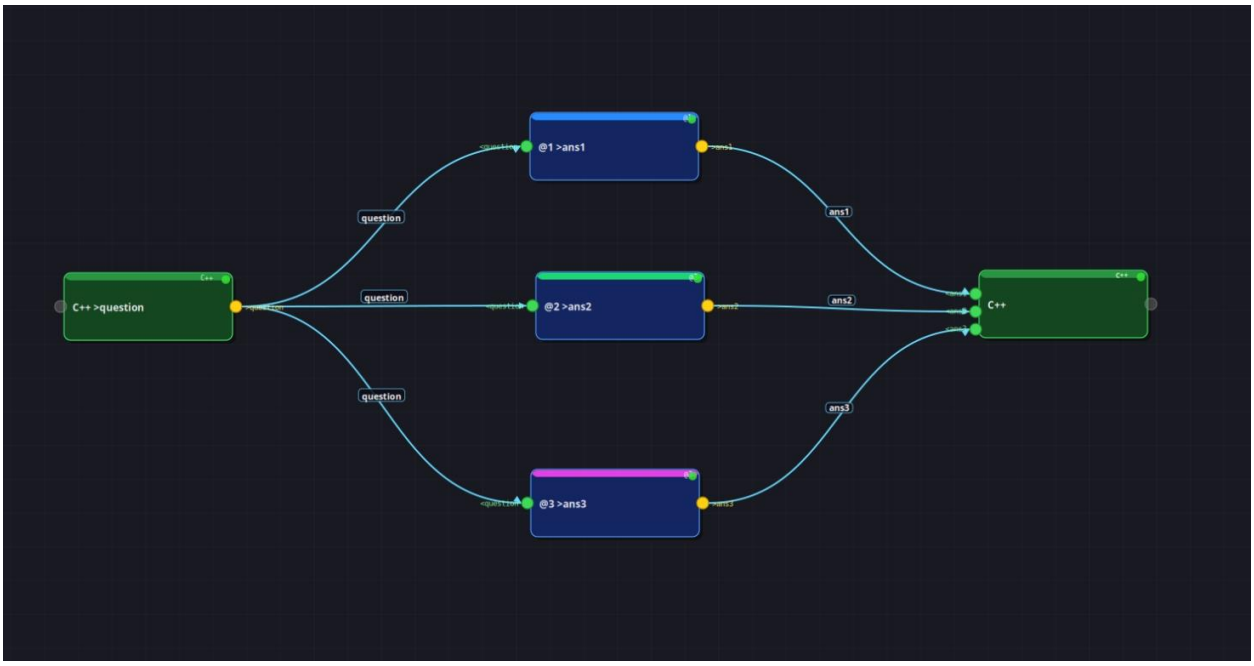
```cpp >question
#include <iostream>
int main() {
    std::cout <<
        "In modern C++17, what is the single most important container or "
        "data structure a developer should deeply understand, and why? "
        "Answer in exactly two sentences."
        << std::endl;
    return 0;
}
```

```cpp <ans1 <ans2 <ans3 #include  #include  #include  // helper: read entire file to string std::string slurp(const char* env_key) { std::ifstream f(std::getenv(env_key)); return std::string((std::istreambuf_iterator(f)), std::istreambuf_iterator()); } int main() { auto a1 = slurp("MSH_VAR_ans1"); auto a2 = slurp("MSH_VAR_ans2"); auto a3 = slurp("MSH_VAR_ans3");
```

```
std::string sep(60, '-');
std::cout << "=== Model @1 (Performance) ===\n" << a1 << "\n"
          << sep << "\n"
          << "=== Model @2 (Practical) ===\n"   << a2 << "\n"
          << sep << "\n"
          << "=== Model @3 (Safety / Modern) ===\n" << a3 << "\n";
return 0;

}
```

Key Patterns

- All three LLM blocks are independent — each reads the original `question` file, writes
  its own output variable.
- The collecting C++ block uses a `slurp()` helper to read multiple variable files cleanly.
- To run them truly in parallel, add `async` to each directive and add an `await=` barrier
  (see Pattern 12).



Pattern 7 — Evaluator-Optimizer Loop

Concept: A generator LLM produces a C++ solution. An evaluator LLM reviews it and
returns exactly `ACCEPTED` or `REJECTED: <reason>`. The loop repeats — passing both the
previous code and the verdict back to the generator — until accepted or the iteration cap
is reached. The final accepted code is compiled and executed by a C++ block.

Use case: Automated iterative refinement of a C++ algorithm until it passes a strict
quality gate (correctness, no raw pointers, must use RAII).

Key concept: `<!--@loop max=N until=verdict:ACCEPTED-->` wraps the
generator-evaluator pair. The loop exits early on the acceptance condition or after N
iterations as a safety cap.

Flow Diagram

cpp >task ↓ [LOOP max=3 until=verdict:ACCEPTED] @1

code (generate / refine C++ code) cpp <code (print generated code) @2 verdict (evaluate: ACCEPTED or REJECTED: reason) cpp <verdict (print verdict) [END_LOOP] ↓ cpp <code (compile final code, run, print output)

Code

```cpp >task
#include <iostream>
int main() {
    std::cout <<
        "Write a C++ function std::vector<int> sieve(int n) that returns all
"
        "prime numbers up to n using the Sieve of Eratosthenes. "
        "Requirements: use RAII (no raw new/delete), mark functions constexpr
"
        "where possible, include a main() that prints primes up to 50."
        << std::endl;
    return 0;
}
```

```cpp <code #include <iostream> #include <fstream> #include <string> int main()
{     std::cout << "=== Generated Code ===\n";     std::ifstream
f(std::getenv("MSH_VAR_code"));     std::string line;     while
(std::getline(f, line)) std::cout << line << "\n";     return 0; }
```

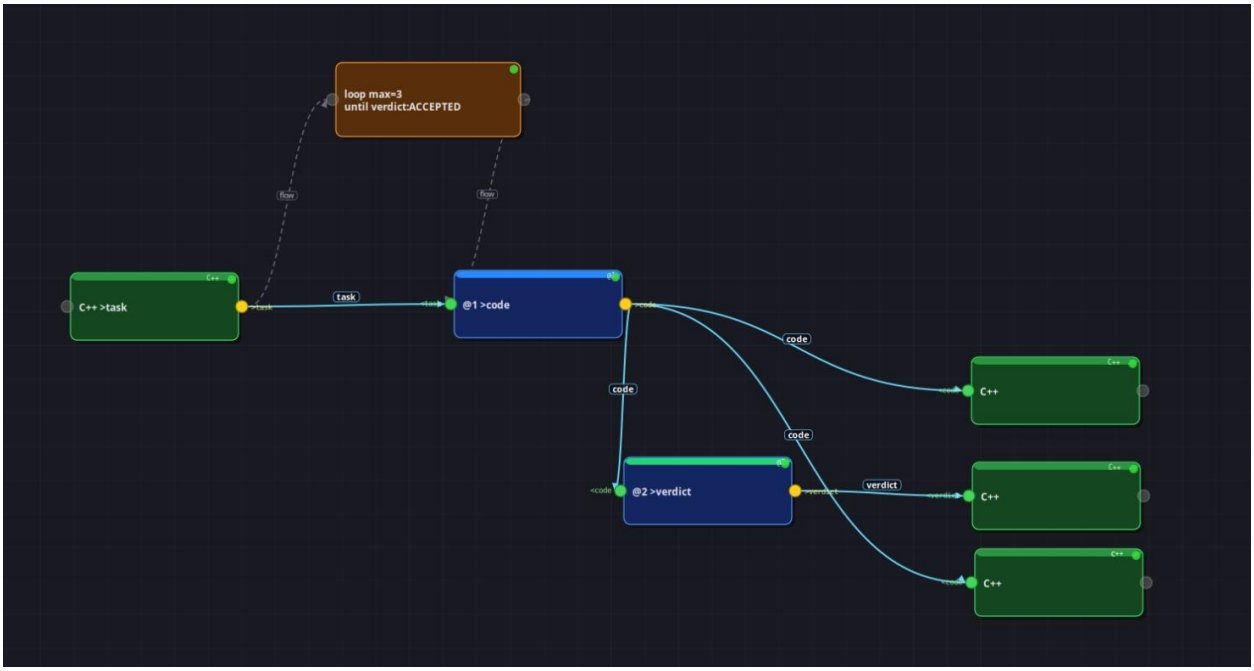
```
cpp <verdict #include <iostream> #include <fstream> #include <sstream>
#include <string> int main() {     std::ifstream
f(std::getenv("MSH_VAR_verdict"));     std::string
verdict((std::istreambuf_iterator<char>(f)),
std::istreambuf_iterator<char>());     // extract first word (ACCEPTED or
REJECTED)     std::istringstream ss(verdict);     std::string first_word;
ss >> first_word;     std::cout << "=== Verdict: " << first_word << " ===\n";
// last stdout line must equal the verdict word for LOOP condition check
std::cout << first_word << "\n";     return 0; }
```

```cpp <code #include  #include  #include  #include  int main() { const char* path = std::getenv("MSH_VAR_code"); std::cout << "=== Final Accepted Code ==="; std::ifstream src(path); std::string line; while (std::getline(src, line)) std::cout << line << ""; src.close();

```
std::cout << "\n=== Compiling & Running ===\n";
// copy to stable .cpp path before compiling
std::string cp_cmd = std::string("cp -- \"") + path + "\" /tmp/msh_p7_gen.cpp
";
std::system(cp_cmd.c_str());
if (std::system("g++ -std=c++17 -O2 -o /tmp/msh_p7_bin /tmp/msh_p7_gen.cpp 2>
&1") != 0) {
    std::cerr << "Compile failed\n"; return 1;
}
std::system("/tmp/msh_p7_bin");
return 0;

}

Key Patterns

- The loop variable `verdict` must contain exactly `ACCEPTED` to trigger earl
y exit —
  prompt the evaluator model very strictly.
- On each loop iteration the generator sees the same `<task` variable, so it
can incorporate
  `REJECTED:` feedback only if you also pass `<verdict` to it (add `<verdict`
to the @1
  directive for self-correction).
- `max=3` is a safety cap — always set it to prevent infinite loops.
```
```



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

Concept: Replaces multi-language pipelines with multiple C++ stages, each acting as a
specialist. C++ generates raw data, a second C++ block computes statistics, Model @1
provides a narrative analysis, Model @2 compresses the analysis to a headline, and a final
C++ block collects everything and formats a structured report.

Use case: Automated benchmark reporting — C++ produces timing data, C++ derives
statistics, two LLMs add interpretation and a tweet-sized summary, C++ formats the final
report.

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

Flow Diagram

cpp >raw_data (generate 12 random latency measurements in ms) ↓ cpp stats (compute mean, median, p95, min, max) ↓ @1 analysis (LLM: narrative interpretation) ↓ @2 headline (LLM: compress to ≤10-word headline) ↓ cpp <raw_data <stats <analysis <headline (print full structured report)

Code

```cpp >raw_data
#include <iostream>
int main() {
    // deterministic pseudo-latencies (ms), comma-separated
    int latencies[] = {12, 45, 8, 102, 23, 67, 15, 88, 34, 201, 19, 56};
    for (int i = 0; i < 12; ++i) {
        std::cout << latencies[i];
        if (i < 11) std::cout << ",";
    }
    std::cout << "\n";
    return 0;
}
```

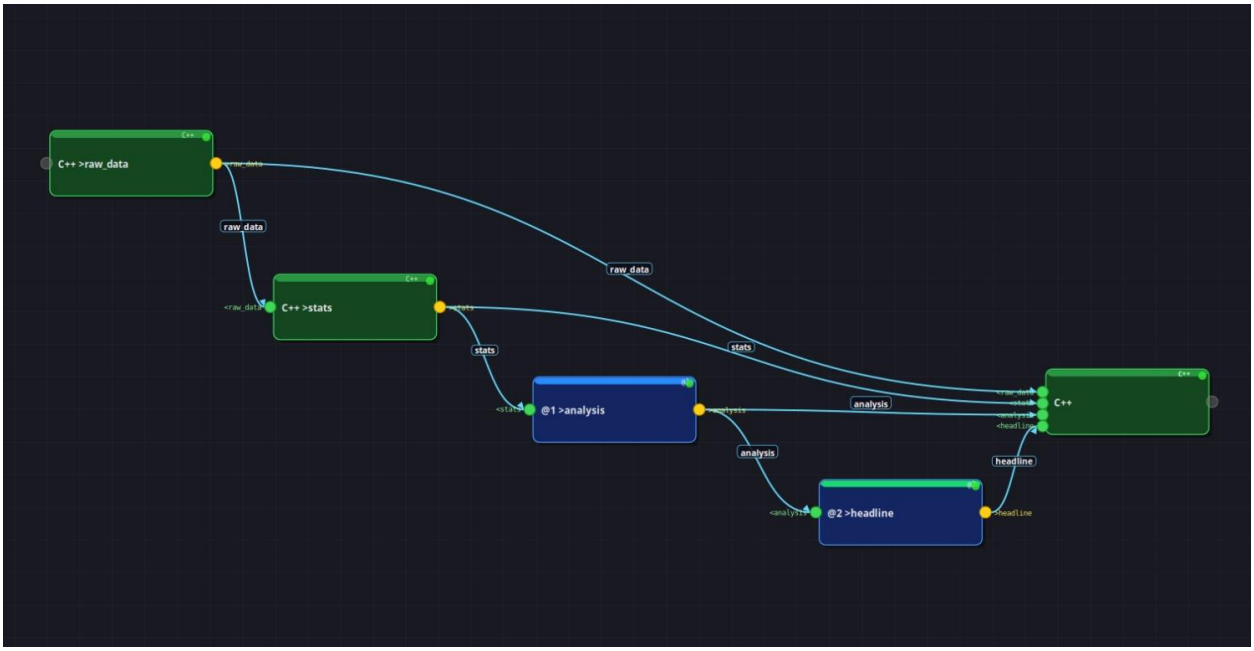
```
cpp <raw_data >stats #include <iostream> #include <fstream> #include <sstream> #include <string> #include <vector> #include <algorithm> #include <numeric> int main() {     std::ifstream f(std::getenv("MSH_VAR_raw_data")); std::string raw;     std::getline(f, raw);     std::vector<double> v; std::stringstream ss(raw);     std::string tok;     while (std::getline(ss, tok, ',')) v.push_back(std::stod(tok));     std::sort(v.begin(), v.end()); double sum   = std::accumulate(v.begin(), v.end(), 0.0);     double mean  = sum / v.size();     double med   = v.size() % 2 == 0                    ? (v[v.size()/2-1] + v[v.size()/2]) / 2.0                    : v[v.size()/2]; double p95   = v[(int)(v.size() * 0.95)];     std::cout << "count=" << v.size()               << " min="  << v.front()               << " max="  << v.back()               << " mean=" << mean               << " median=" << med << " p95="  << p95               << "\n";     return 0; }
```

```cpp <raw_data <stats <analysis <headline #include  #include  #include  std::string slurp(const char* key) { std::ifstream f(std::getenv(key)); return {std::istreambuf_iterator(f), {}}; } int main() { auto raw = slurp("MSH_VAR_raw_data"); auto stats = slurp("MSH_VAR_stats"); auto analysis = slurp("MSH_VAR_analysis"); auto headline = slurp("MSH_VAR_headline");

std::string sep(60, '=');
std::cout << sep << "\n"
          << "BENCHMARK REPORT\n"
          << sep << "\n"
          << "Raw data (ms) : " << raw
          << "Statistics    : " << stats
          << "\nAnalysis:\n"    << analysis
```

```
          << "\nHeadline: >>> " << headline << " <<<\n"
          << sep << "\n";
return 0;

}
```

Key Patterns

- C++ replaces every other language: data generation, statistics, and report formatting
  are all separate compiled binaries communicating via mshell variables.
- `slurp()` helper avoids repetitive file-reading boilerplate when a block needs many
  input variables.
- LLM directives slot in between C++ stages just like any other block.



Pattern 9 Routing (LLM Classifies → Conditional C++ Branch Executes)

Concept: An LLM classifies a C++-generated input into one of several categories. Each
downstream C++ block carries an `if=route:VALUE` condition — only the matching block
runs; all others are silently skipped.

Use case: Smart computation dispatcher — input can be a sorting request,

a math
formula, or a text transformation; the LLM routes it to the correct C++ handl
er automatically.

Key concept: `if=varname:expected_value` on any block makes it conditiona
l. The LLM
classifier must return a single, predictable word — prompt discipline is crit
ical.

Flow Diagram

cpp >input ↓ @1 route (classify: SORT / MATH / TEXT) ↓ cpp <input if=route:SORT (run sort algorithm, print sorted result) cpp <input if=route:MATH (evaluate the math expression, print result) cpp <input if=route:TEXT (reverse & uppercase the string, print) ↓ cpp <route (print which branch was taken)

Three separate test runs are shown (one per category).

Code — Test 1: SORT

```cpp >input1
#include <iostream>
int main() {
    std::cout << "sort these integers: 9 3 7 1 5 8 2 6 4" << std::endl;
    return 0;
}
```

```
cpp <input1 if=route1:SORT // SORT branch: extract integers and sort them
#include <iostream> #include <fstream> #include <sstream> #include <string>
#include <vector> #include <algorithm> int main() {     std::ifstream
f(std::getenv("MSH_VAR_input1"));     std::string line; std::getline(f,
line);     // extract digits from the sentence     std::istringstream
ss(line);     std::string tok;     std::vector<int> nums;     while (ss >>
tok) {         try { nums.push_back(std::stoi(tok)); } catch(...) {}     }
std::sort(nums.begin(), nums.end());     std::cout << "[SORT branch] sorted:
";     for (int n : nums) std::cout << n << " ";     std::cout << "\n";
return 0; }
```

```
cpp <input1 if=route1:MATH #include <iostream> int main() { std::cout <<
"[MATH branch — unexpected for test 1]\n"; return 0; }
```

```
cpp <input1 if=route1:TEXT #include <iostream> int main() { std::cout <<
"[TEXT branch — unexpected for test 1]\n"; return 0; }
```

```
cpp <route1 #include <iostream> #include <fstream> #include <string> int main() {     std::ifstream f(std::getenv("MSH_VAR_route1"));     std::string r; std::getline(f, r);     std::cout << "Test1 routed to: " << r << "\n\n"; return 0; }
```

Code — Test 2: MATH

```
cpp >input2 #include <iostream> int main() {     std::cout << "compute the value of: 2 to the power of 10" << std::endl;     return 0; }

cpp <input2 if=route2:SORT #include <iostream> int main() { std::cout << "[SORT branch — unexpected for test 2]\n"; return 0; }

cpp <input2 if=route2:MATH // MATH branch: compute 2^10 #include <iostream> #include <cmath> int main() {     long long result = (long long)std::pow(2, 10);     std::cout << "[MATH branch] 2^10 = " << result << "\n";     return 0; }

cpp <input2 if=route2:TEXT #include <iostream> int main() { std::cout << "[TEXT branch — unexpected for test 2]\n"; return 0; }

cpp <route2 #include <iostream> #include <fstream> #include <string> int main() {     std::ifstream f(std::getenv("MSH_VAR_route2"));     std::string r; std::getline(f, r);     std::cout << "Test2 routed to: " << r << "\n\n"; return 0; }
```

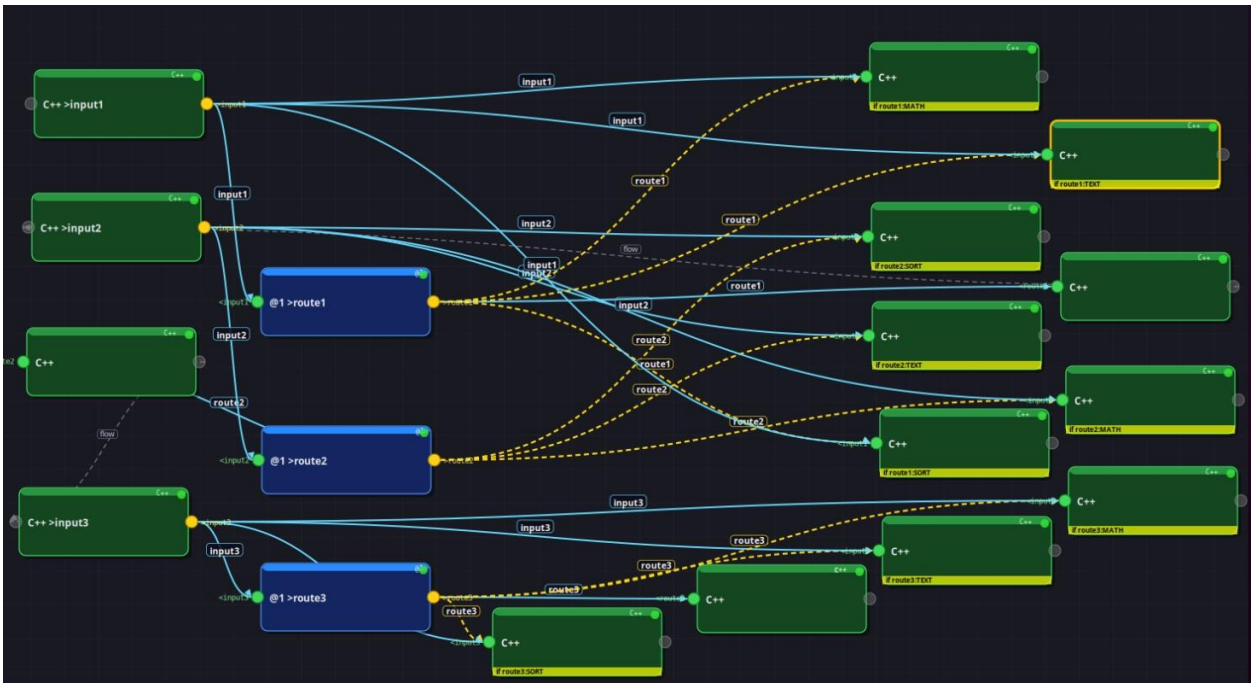
Code — Test 3: TEXT

```
cpp >input3 #include <iostream> int main() {     std::cout << "transform this string: hello mshell world" << std::endl;     return 0; }

cpp <input3 if=route3:SORT #include <iostream> int main() { std::cout << "[SORT branch — unexpected for test 3]\n"; return 0; }

cpp <input3 if=route3:MATH #include <iostream> int main() { std::cout << "[MATH branch — unexpected for test 3]\n"; return 0; }

cpp <input3 if=route3:TEXT // TEXT branch: reverse & uppercase the input string #include <iostream> #include <fstream> #include <string> #include <algorithm> #include <cctype> int main() {     std::ifstream f(std::getenv("MSH_VAR_input3"));     std::string line; std::getline(f, line);     // extract the substring after ": "     auto pos = line.find(": ");     std::string s = (pos != std::string::npos) ? line.substr(pos + 2) : line;     std::reverse(s.begin(), s.end());     for (char& c : s) c = std::toupper(c);     std::cout << "[TEXT branch] reversed+upper: " << s << "\n";     return 0; }

cpp <route3 #include <iostream> #include <fstream> #include <string> int main() {     std::ifstream f(std::getenv("MSH_VAR_route3"));     std::string
```

```
r; std::getline(f, r);     std::cout << "Test3 routed to: " << r << "\n";
return 0; }
```

Key Patterns

- `if=route:SORT` — the block runs only when the variable `route` contains exactly `SORT`.
- Skipped blocks produce no output and no variable writes — as if they don't exist.
- Always include "else" stubs (with unexpected-branch messages) during development to catch mis-classifications.



Pattern 10 — Full C++ Pipeline (All Patterns Combined)

Concept: A complete end-to-end showcase: C++ generates prime numbers, a second C++ block derives statistics, two LLMs analyse and poeticise in parallel (async), a C++ `await=` barrier synchronises them, a third LLM synthesises the results, and a final C++ block renders the output in a decorative frame.

Use case: Reference template — demonstrates that all building blocks (sequential stages, fan-out, async LLM, await barrier, synthesis) compose naturally in a single mshell document using only C++.

Key concept: There is no special "combine patterns" syntax. Every block simply reads named variables — the document structure IS the execution graph.

Flow Diagram

```
cpp >raw_data           (Sieve: first 10 primes)
  ↓
cpp <raw_data >stats    (min, max, mean, sum)
  ↓         ↓
@1 <stats >analysis async    @2 <raw_data >poem async
  ↓         ↓
cpp await=analysis,poem
  ↓
cpp <analysis <poem     (print both)
  ↓
@1 <analysis <poem >combined   (synthesise into one elegant sentence)
  ↓
cpp <combined           (render in decorative C++ frame)
```

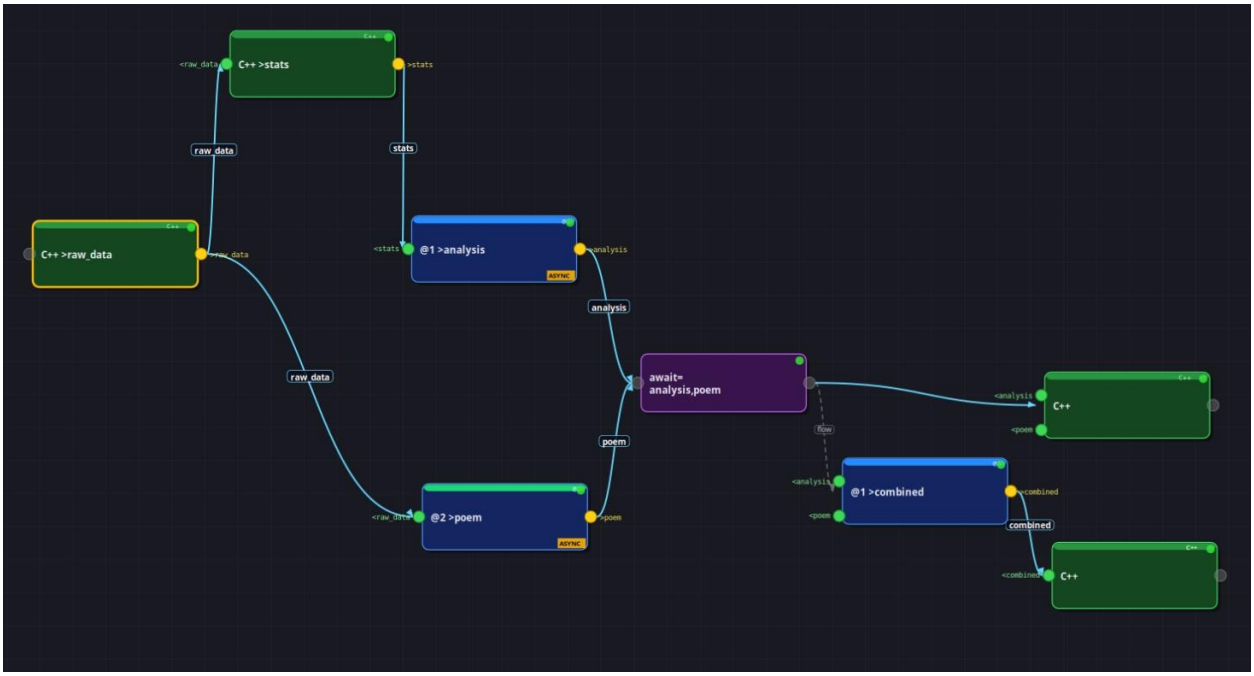
Code

```
cpp >raw_data #include <iostream> #include <vector> int main() {
std::vector<int> primes;     for (int n = 2; (int)primes.size() < 10; ++n) {
bool ok = true;         for (int i = 2; i * i <= n; ++i) if (n % i == 0) { ok
= false; break; }         if (ok) primes.push_back(n);     }     for (int i =
0; i < (int)primes.size(); ++i) {         std::cout << primes[i];         if
(i + 1 < (int)primes.size()) std::cout << " ";     }     std::cout << "\n";
return 0; }
```

```
cpp <raw_data >stats #include <iostream> #include <fstream> #include
<sstream> #include <string> #include <vector> #include <numeric> #include
<algorithm> int main() {     std::ifstream
f(std::getenv("MSH_VAR_raw_data"));     std::string raw; std::getline(f,
raw);     std::istringstream ss(raw);     std::vector<int> v;     int x;
while (ss >> x) v.push_back(x);     double mean = std::accumulate(v.begin(),
v.end(), 0.0) / v.size();     std::cout << "count=" << v.size()
<< " min="  << *std::min_element(v.begin(), v.end())              << " max="
<< *std::max_element(v.begin(), v.end())              << " sum="  <<
std::accumulate(v.begin(), v.end(), 0)              << " mean=" << mean
<< "\n";     return 0; }
```

```
cpp await=analysis,poem // synchronisation barrier — no code needed, just the
await= attribute #include <iostream> int main() {     std::cout << "[await]
both LLM jobs complete\n";     return 0; }
```

```
cpp <analysis <poem #include <iostream> #include <fstream> #include <string>
std::string slurp(const char* key) {     std::ifstream f(std::getenv(key));
return {std::istreambuf_iterator<char>(f), {}}; } int main() {     std::cout
<< "=== Analysis ===\n" << slurp("MSH_VAR_analysis") << "\n"              <<
"=== Poem ===\n"     << slurp("MSH_VAR_poem")     << "\n";     return 0; }
```

```cpp <combined #include  #include  #include  int main() { std::ifstream f(std::getenv("MSH_VAR_combined")); std::string text((std::istreambuf_iterator(f)), {}); // trim trailing newline while (!text.empty() && text.back() == '') text.pop_back();

```
std::string border(text.size() + 4, '*');
std::cout << border << "\n"
          << "* " << text << " *\n"
          << border << "\n";
return 0;
```

}

---

## Key Patterns

- `async` on an LLM directive launches a background job; `await=analysis,poem` blocks
  until both output variables are written.
- Total time = max(time(@1), time(@2)), not the sum.
- The synthesising LLM (`@1 <analysis <poem`) receives both variables with
  `[varname]:` labels injected automatically.
- No special "combine" syntax — variable names are the only wiring mechanism.




---

## Pattern 11 — MShell Node with Multiple Models (C++ Data Sources)
```

Concept: Native `mshell` code blocks call LLM models using `ollama1`/`oll
ama2`
commands, with C++ blocks producing and consuming the surrounding variables.

Use case: Workflows where mshell's native AI commands are preferable to L
LM
directives — for example, when you need to build the prompt dynamically insid
e the mshell
block itself, or combine mshell commands (`print`, `eval`) with model calls.

Key concept: `mshell` blocks support the same `<in_vars` and `>out_var` v
ariable
system as C++ blocks. Input variables are loaded into the mshell variable tab
le before
execution, and stdout is captured into the output variable.

Flow Diagram

cpp >topic (produce topic string) cpp >constraints (produce constraint string) ↓ mshell <topic explanation (ollama1: explain topic) ↓ mshell keywords (ollama2: extract keywords) ↓ cpp <topic <explanation <keywords (format final report)

Code

```cpp >topic
#include <iostream>
int main() {
    std::cout << "move semantics and rvalue references in C++11" << std::endl
;
    return 0;
}
```

```
cpp >constraints #include <iostream> int main() {     std::cout << "beginner-
friendly, max 2 sentences, include a one-line analogy" << std::endl;
return 0; }
```

```
mshell <topic <constraints >explanation ollama1 "Explain $topic. Style:
$constraints"
```

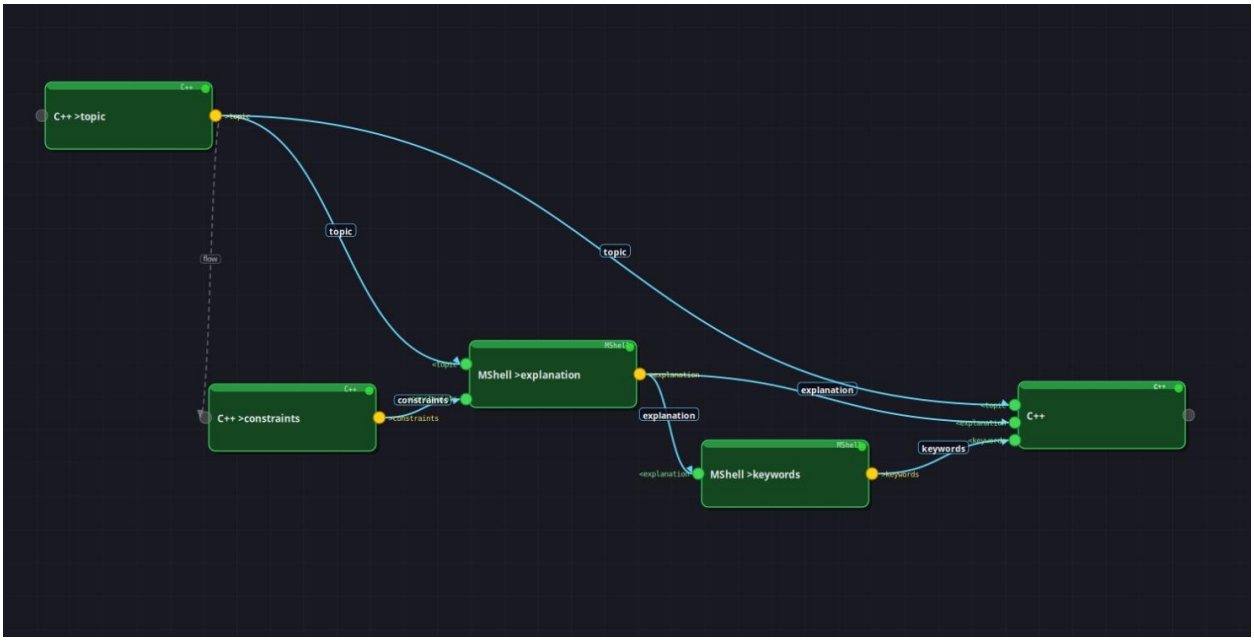
```
mshell <explanation >keywords ollama2 "Extract exactly 4 technical keywords
from this text: $explanation. Reply with only the 4 words, comma-separated."
```

```
cpp <topic <explanation <keywords #include <iostream> #include <fstream>
#include <string> std::string slurp(const char* key) {     std::ifstream
```

```
f(std::getenv(key));     return {std::istreambuf_iterator<char>(f), {}}; }
int main() {     std::cout << "=== Topic ===\n"       <<
slurp("MSH_VAR_topic")               << "\n=== Explanation ===\n" <<
slurp("MSH_VAR_explanation")               << "\n=== Keywords ===\n"    <<
slurp("MSH_VAR_keywords") << "\n";     return 0; }
```

Key Patterns

- `mshell` blocks interpolate variable contents with `$varname` syntax directly in the `ollama1`/`ollama2` command string.
- C++ blocks handle structured data production and final formatting; mshell blocks handle the AI calls.
- `ollama1exec`, `ollama2exec`, `ollama3exec` are internal implementations for running LLM-generated code directly — use them when the model output is itself executable.



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

Concept: Three LLM models are launched asynchronously in parallel, each with a different instructional angle on the same C++-generated question. An `await=` barrier blocks until all three finish. A fourth LLM synthesises the three perspectives, and a C++ block renders the final output.

Use case: High-quality C++ concept explanation — collect a beginner explanation, a memory-model deep dive, and a real-world analogy simultaneously, then synthesise the best of all three. Async execution reduces total wall-clock time to the slowest model.

Key concept: `async` on an LLM directive launches it as a background process. `cpp await=ans1,ans2,ans3` blocks until all named variables are written. The synthesising LLM receives all three via `<ans1 <ans2 <ans3`.

Flow Diagram

```
cpp >question
  ↓           ↓              ↓
@1 <question  @2 <question   @3 <question
>ans1 async   >ans2 async    >ans3 async
  ↓           ↓              ↓
cpp await=ans1,ans2,ans3
  ↓
@1 <ans1 <ans2 <ans3 >final  (synthesise)
  ↓
cpp <final                   (render result)
```

Code

```
cpp >question #include <iostream> int main() {     std::cout <<         "What
is std::move() in C++11 and why does it matter for performance?"         <<
std::endl;     return 0; }
```

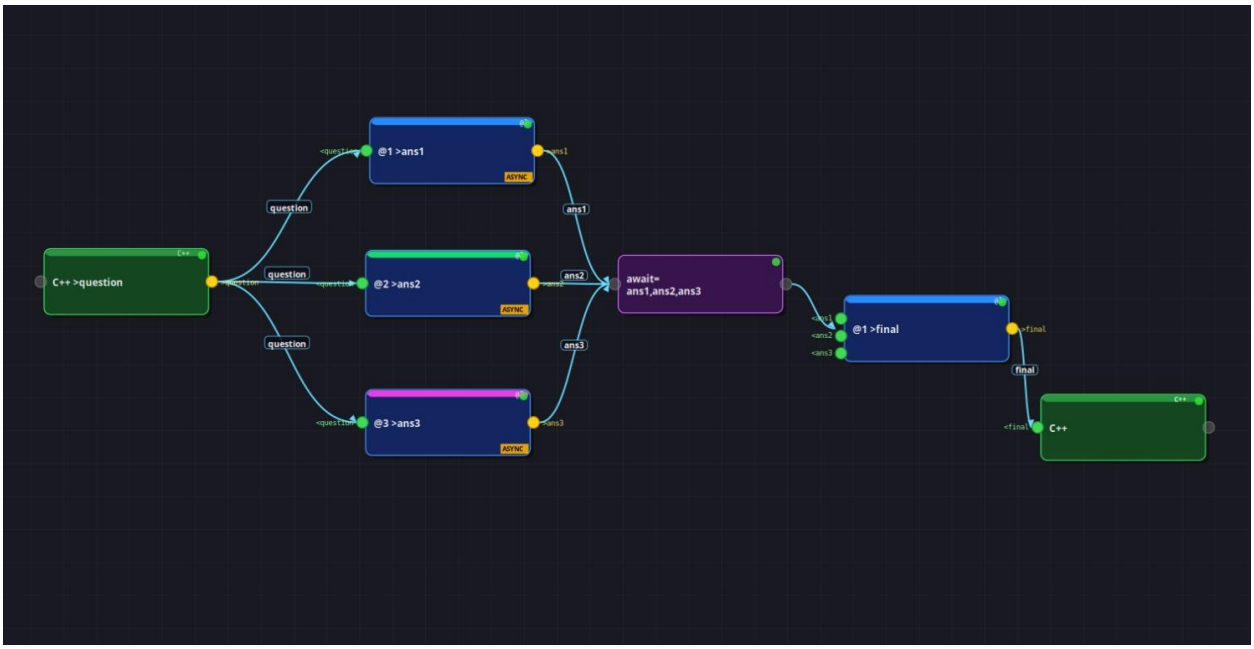
```
cpp await=ans1,ans2,ans3 // await barrier: wait for all three async LLM jobs
#include <iostream> int main() {     std::cout << "[await] all three LLM
responses received\n";     return 0; }
```

```cpp <final #include  #include  #include  std::string slurp(const char* key) { std::ifstream f(std::getenv(key)); return {std::istreambuf_iterator(f), {}}; } int main() { auto a1 = slurp("MSH_VAR_ans1"); auto a2 = slurp("MSH_VAR_ans2"); auto a3 = slurp("MSH_VAR_ans3"); auto fin = slurp("MSH_VAR_final");

```
std::string sep(60, '-');
std::cout << "[Beginner]   " << a1
          << sep << "\n"
          << "[Memory model] " << a2
          << sep << "\n"
          << "[Analogy]   " << a3
          << sep << "\n"
          << ">>> Synthesis: " << fin << "\n";
return 0;

}
```
```

Key Patterns

- `async` jobs run in parallel background processes — total time ≈ slowest model, not the
  sum of all three.
- `await=ans1,ans2,ans3` must list every async output variable before proceeding.
- When the synthesising LLM receives multiple `<invar`, mshell injects each with a
  `[varname]:` label — reference them explicitly in the prompt for clarity.
- The `cpp await=` block must still compile and run; an empty `main()` that prints a
  status message is the recommended pattern.



Part II — Patterns 13–24: Advanced Node Types

Pattern 13 — WHILE Loop: Iterative Counter with LLM Commentary

Concept: A C++ block initialises a counter and a status flag. The WHILE loop runs
while `status == running`. Each iteration a C++ block increments the counter and writes
both values directly to their `$MSH_VAR_*` files. When the counter reaches th

e target the
status is set to `done`, stopping the loop. An LLM generates one interesting fact about the
current count. A C++ block prints the iteration result.

Use case: Iterative computation log — count through a sequence, annotate each step
with LLM commentary, stop at a threshold.

Key concept: A C++ block with multiple `>outvar` must write to each `$MSH_VAR_*` file directly via `std::ofstream` — mshell does not capture stdout when
multiple output variables are declared.

Flow Diagram

cpp >status="running" cpp >counter="0" ↓ [WHILE status:running] cpp <counter counter >status (increment; write via MSH_VAR_*) @1 comment (LLM: one interesting fact) cpp <comment (print iteration result) [END_WHILE] ↓ cpp <counter (final output)

Code

```cpp >status
#include <iostream>
int main() { std::cout << "running\n"; return 0; }
```

```
cpp >counter #include <iostream> int main() { std::cout << "0\n"; return 0; }
```

```
cpp <counter <status >counter >status #include <iostream> #include <fstream> #include <string> int main() {     // read current counter     std::ifstream fc(std::getenv("MSH_VAR_counter"));     int val; fc >> val; fc.close();     ++val;     // write new counter     std::ofstream wc(std::getenv("MSH_VAR_counter"));     wc << val << "\n"; wc.close();     // write new status     std::ofstream ws(std::getenv("MSH_VAR_status"));     ws << (val >= 4 ? "done" : "running") << "\n"; ws.close();     // stdout NOT captured (multiple outvars) — this just goes to terminal log     std::cout << "[cpp] counter=" << val << "\n";     return 0; }
```

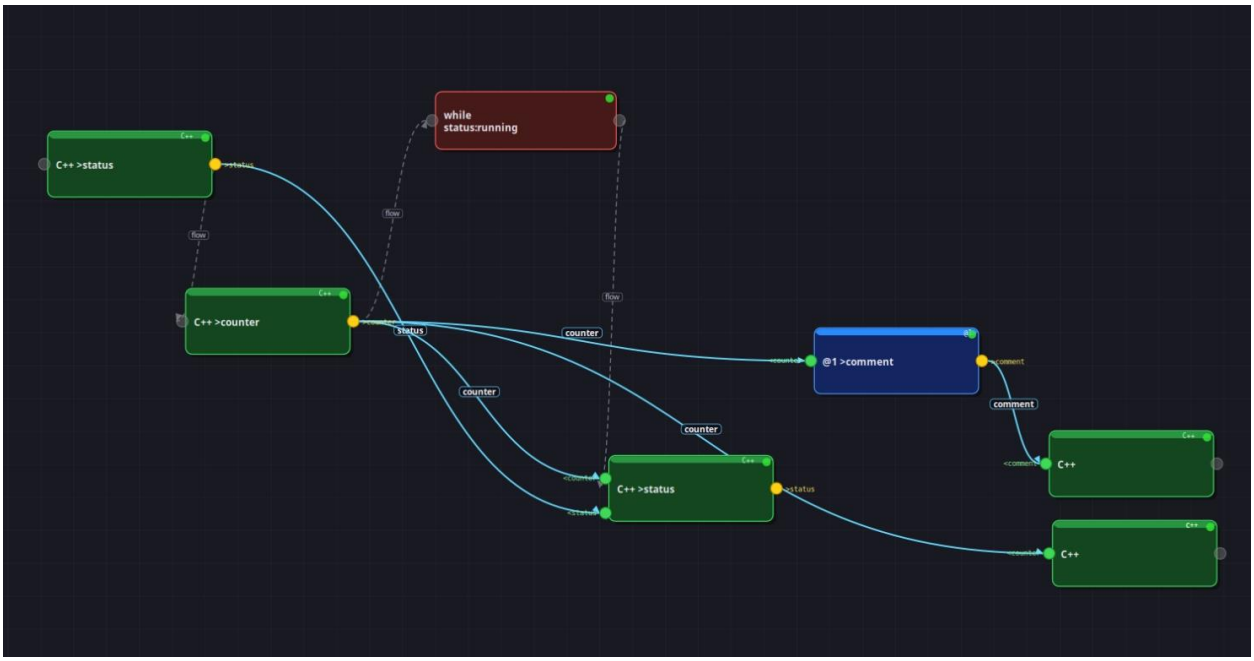
```
cpp <comment #include <iostream> #include <fstream> #include <string> int main() {     std::ifstream fc(std::getenv("MSH_VAR_counter"));     int val; fc >> val; fc.close();     std::ifstream fm(std::getenv("MSH_VAR_comment"));     std::string comment((std::istreambuf_iterator<char>(fm)), {});     std::cout << "[iter " << val << "] " << comment << "\n";     return 0; }
```

```
cpp <counter #include <iostream> #include <fstream> int main() {     std::ifstream f(std::getenv("MSH_VAR_counter"));     int val; f >> val;
```

```
std::cout << "=== WHILE done. Final counter = " << val << " ===\n";
return 0; }
```

Key Patterns

- **Multiple `>outvar` on a C++ block:** use `std::ofstream` to write directly to each `std::getenv("MSH_VAR_*")` path — mshell does NOT capture stdout in this case.
- WHILE reads the **last non-empty line** of the condition variable.
- Use a `running`/`done` flag as the exit condition — the most reliable pattern.
- All variables (`status`, `counter`) must be initialised before the loop because WHILE checks the condition before the first iteration.



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

Concept: A C++ block creates a newline-separated list of items. FOREACH iterates over the list line by line. On each iteration the runtime automatically sets the iterator variable to the current item. An LLM generates a response for each item. A C++ block prints the result.

Use case: Automated C++ concept glossary — iterate over a list of C++ features, ask the LLM for a one-sentence definition of each, print the mini-glossary.

Key concept: The list must be created with one item per line (`\n`-separated). The iterator variable is set automatically by the runtime before each iteration.

Flow Diagram

```
cpp >features           (newline-separated list of C++ features)
  ↓
[FOREACH feature in features]
  @1 <feature >definition   (LLM: one-sentence definition)
  cpp <definition            (print feature name + definition)
[END_FOREACH]
  ↓
cpp                     (completion message)
```

Code

```
cpp >features #include <iostream> int main() {     // one feature per line —
mandatory for FOREACH     std::cout <<
"RAII\nstd::optional\nstd::variant\nMove semantics\nConcepts (C++20)" <<
std::endl;     return 0; }
```

```cpp <definition #include  #include  #include  int main() { std::ifstream
ff(std::getenv("MSH_VAR_feature")); std::string feature((std::istreambuf_iterator(ff)), {});
while (!feature.empty() && feature.back() == '') feature.pop_back();

std::ifstream fd(std::getenv("MSH_VAR_definition"));
std::string def((std::istreambuf_iterator<char>(fd)), {});

std::cout << "--- " << feature << " ---\n" << def << "\n\n";
return 0;

}

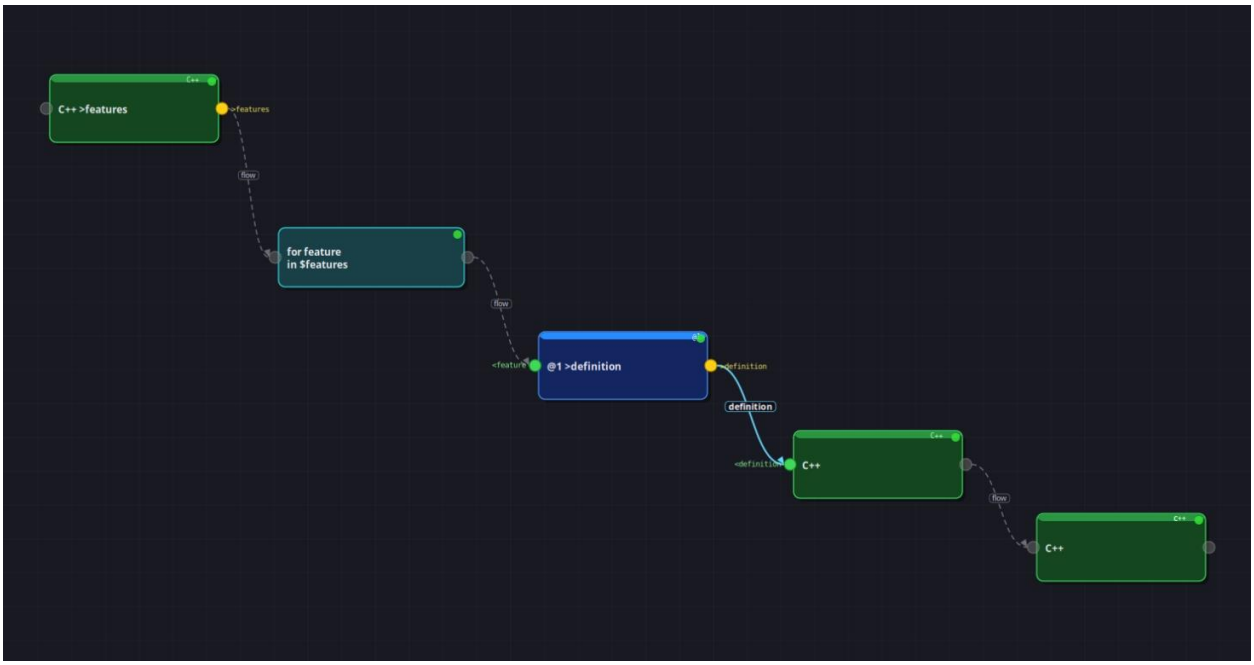

<!--@end_foreach-->

```cpp
#include <iostream>
int main() {
    std::cout << "=== C++ Feature Glossary complete ===\n";
    return 0;
}
```

Key Patterns

- The C++ block producing the list must output **one item per line** — space-separated output will not split correctly.
- `MSH_VAR_feature` is automatically populated by the FOREACH runtime before each C++ block runs.
```

- Any number of C++ and LLM blocks can appear inside a FOREACH body.
- Trim trailing `\n` from iterator variables with `while (!s.empty() && s.back() == '\n') s.pop_back();`.



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

Concept: A C++ block initialises an input. The TRY block wraps a C++ block that intentionally triggers a runtime error (divide-by-zero detected at runtime, exits with `exit(1)`). Because the block exits with a non-zero code the parser skips the rest of the TRY body and jumps to CATCH. The CATCH block prints a hardcoded error message. After the TRY/CATCH a safe fallback C++ block processes the input normally.

Use case: Robust pipeline — attempt a risky computation (e.g. parsing untrusted data, calling an external binary), catch failure gracefully, continue with a safe fallback.

Key concept: - Exit with a non-zero code (via `exit(1)` or `return 1`) to trigger CATCH. - CATCH block does **not** use `<errvar` — print the literal string `"try_block_failed"` directly. - Pipeline continues normally after `<!--@end_try-->`.

Flow Diagram

```
cpp >input              (set up input data)
  ↓
[TRY]
```

```
  cpp <input >result    (risky computation — may exit(1))
[CATCH >error]
  cpp                   (print literal "try_block_failed" message)
[END_TRY]
  ↓
cpp <input >safe_result (safe fallback computation)
cpp <safe_result        (print result)
```

Code

```
cpp >input #include <iostream> int main() {     // A CSV line of integers —
one of which is zero (will cause division issues)     std::cout <<
"100,25,0,80,50\n";     return 0; }
```

```cpp result #include  #include  #include  #include  #include  #include  int main() {
std::ifstream f(std::getenv("MSH_VAR_input")); std::string line; std::getline(f, line);
std::istringstream ss(line); std::string tok; std::vector nums; while (std::getline(ss, tok, ','))
nums.push_back(std::stoi(tok));

// intentional risky operation: divide first value by each subsequent value
// will fail when divisor == 0
for (int i = 1; i < (int)nums.size(); ++i) {
    if (nums[i] == 0) {
        std::cerr << "ERROR: division by zero detected at index " << i << "\n
";
        std::exit(1);   // triggers CATCH
    }
    std::cout << nums[0] << " / " << nums[i] << " = " << nums[0] / nums[i] <<
"\n";
}
return 0;

}


<!--@catch >error-->

```cpp
#include <iostream>
int main() {
    std::cout << "=== Caught error: try_block_failed ===\n"
              << "Division pipeline aborted. Switching to safe fallback.\n";
    return 0;
}
```

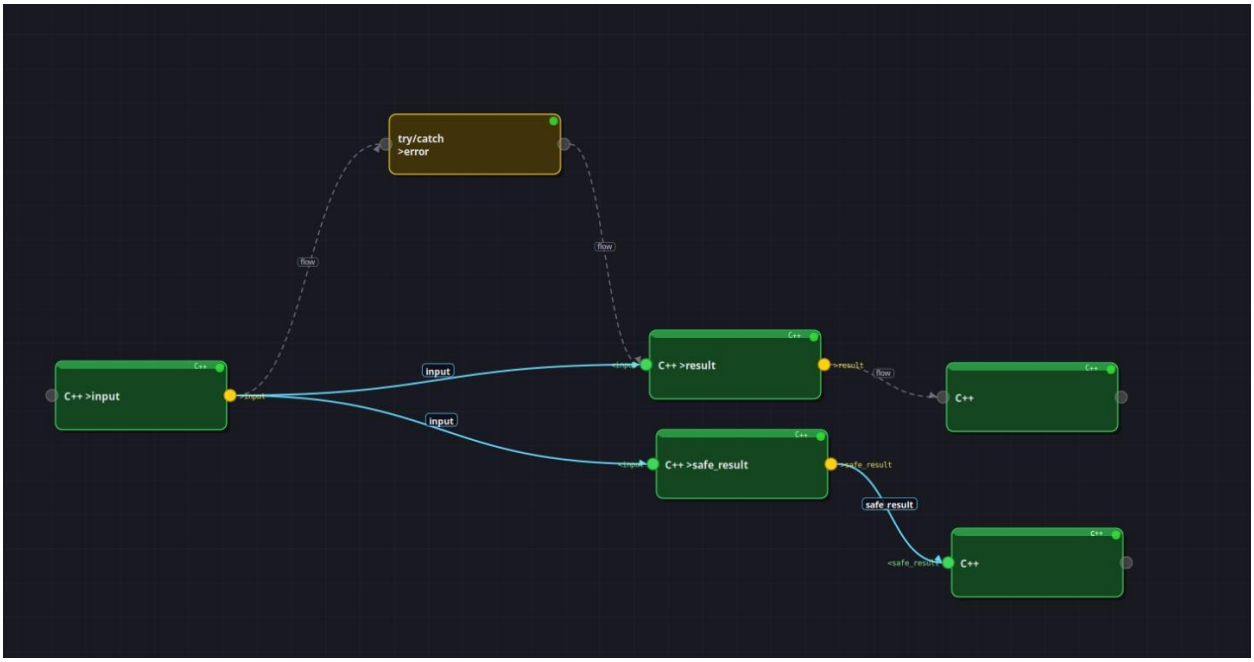
cpp <input >safe_result #include <iostream> #include <fstream> #include
<sstream> #include <string> #include <vector> #include <numeric> int main() {
std::ifstream f(std::getenv("MSH_VAR_input"));   std::string line;
std::getline(f, line);   std::istringstream ss(line);   std::string tok;
```

```
std::vector<int> nums;     while (std::getline(ss, tok, ','))
nums.push_back(std::stoi(tok));     // safe: just compute sum and count (no
division)     int sum = std::accumulate(nums.begin(), nums.end(), 0);
std::cout << "Safe fallback: count=" << nums.size()               << " sum="
<< sum               << " mean=" << (double)sum / nums.size()
<< "\n";     return 0; }
```

```
cpp <safe_result #include <iostream> #include <fstream> #include <string> int
main() {     std::ifstream f(std::getenv("MSH_VAR_safe_result"));
std::string line;     std::cout << "=== Safe Result ===\n";     while
(std::getline(f, line)) std::cout << line << "\n";     return 0; }
```

Key Patterns

- `std::exit(1)` (or `return 1` from `main()`) guarantees a non-zero exit code on any error.
- CATCH block must **not** use `<error` — print the literal string `"try_block_failed"` directly.
- The first block inside TRY to fail skips all remaining TRY blocks; control jumps to CATCH.
- After `<!--@end_try-->` the pipeline continues normally regardless of whether TRY or CATCH ran.



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

Concept: A C++ block creates a two-line dataset. SPLIT divides it by lines into `dataset_1` and `dataset_2`. Two async LLM calls analyse each half in parallel. An `await=` barrier synchronises them. MERGE marks the reduction point. A single LLM synthesises both analyses. A final C++ block prints everything.

Use case: Parallel analysis of two separate benchmark result sets — split the data, analyse each independently, merge into a unified performance report.

Key concept: - 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, not the sum.

Flow Diagram

```
cpp >dataset            ("set_a values\nset_b values")
  ↓
[SPLIT dataset into 2]  → dataset_1, dataset_2
  @1 <dataset_1 >analysis1  async ─┐
  @2 <dataset_2 >analysis2  async ─┤
  cpp await=analysis1,analysis2  ──┘
[MERGE]
  @1 <analysis1 <analysis2 >combined   (synthesise)
  ↓
cpp <combined           (print full report)
```

Code

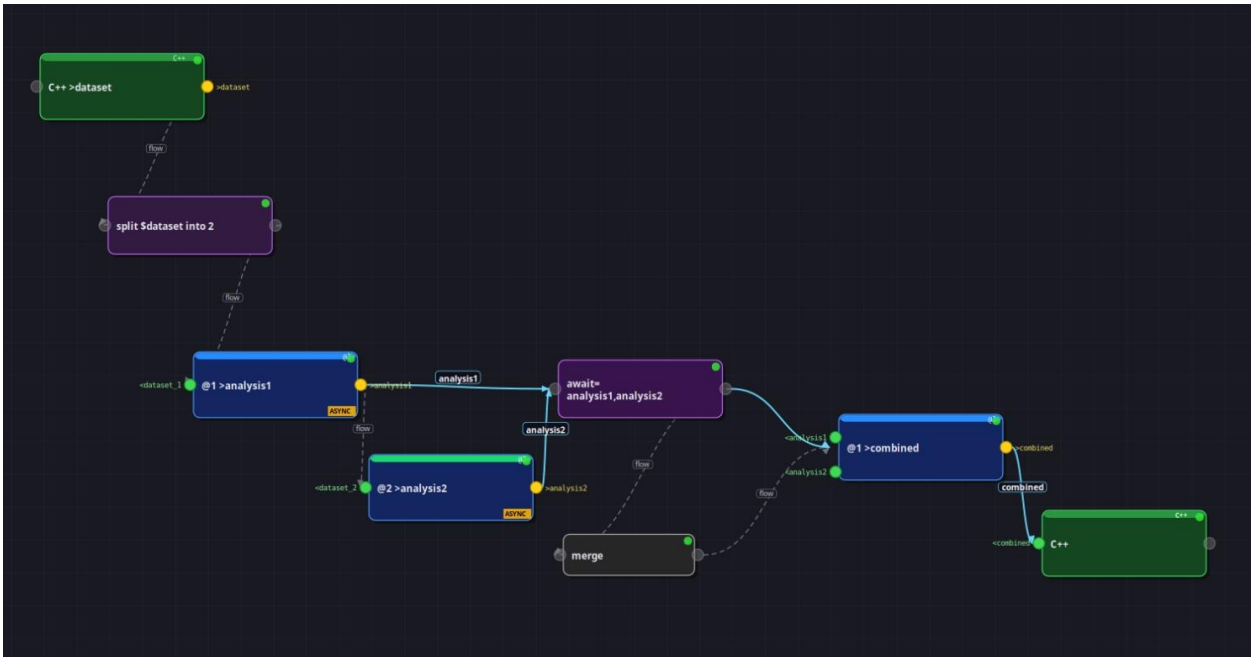
```
cpp >dataset #include <iostream> int main() {     // two lines → SPLIT will
create dataset_1 and dataset_2     std::cout << "benchmark_A: 12ms 15ms 11ms
14ms 16ms 13ms\n"               << "benchmark_B: 45ms 42ms 48ms 44ms 46ms
43ms\n";     return 0; }
```

```
cpp await=analysis1,analysis2 #include <iostream> int main() {     std::cout
<< "[await] both analyses ready\n";     return 0; }
```

```
cpp <combined #include <iostream> #include <fstream> #include <string>
std::string slurp(const char* key) {     std::ifstream f(std::getenv(key));
return {std::istreambuf_iterator<char>(f), {}}; } int main() {     std::cout
<< "=== Benchmark A ===\n" << slurp("MSH_VAR_dataset_1")               <<
"Analysis A: "          << slurp("MSH_VAR_analysis1") << "\n"
<< "=== Benchmark B ===\n" << slurp("MSH_VAR_dataset_2")               <<
"Analysis B: "          << slurp("MSH_VAR_analysis2") << "\n"
<< "=== Merged Report ===\n" << slurp("MSH_VAR_combined") << "\n";     return
0; }
```

Key Patterns

- SPLIT reads `dataset` line-by-line and writes `dataset_1` (line 1), `dataset_2` (line 2), etc.
- MERGE is a **visual-only** marker — it generates no code and writes no variables.
- An `await=` barrier **must** appear before MERGE when async jobs are in flight.
- The reduction LLM receives both analyses with `[varname]:` labels — reference them in the prompt for clarity.



Pattern 17 — CONFIG Node: Parameterized Pipeline

Concept: A CONFIG block documents the pipeline parameters. C++ blocks initialise the actual runtime variables via `std::cout`. Two LLMs generate and post-process content based on those parameters. A C++ block reads all variables and prints a formatted report.

Use case: Reusable documentation generator — change `topic` and `audience` in two C++ lines to produce tailored C++ concept explanations for any subject and reader level.

Key concept: - CONFIG is a **documentation block** — it does NOT inject variables at runtime. Always pair with C++ blocks that `std::cout` the actual values. - `msh_run_and_capture` prints stdout to screen AND writes to the variable — avoid duplicating output with an extra print block.

Flow Diagram

```
config (topic, audience, max_sentences — documentation only)
cpp >topic = "smart pointers in C++"
cpp >audience = "junior C++ developer"
  ↓
@1 <topic <audience >explanation      (generate explanation)
@2 <explanation >keywords             (extract keywords)
cpp <explanation <keywords >report    (print formatted report)
```

Code

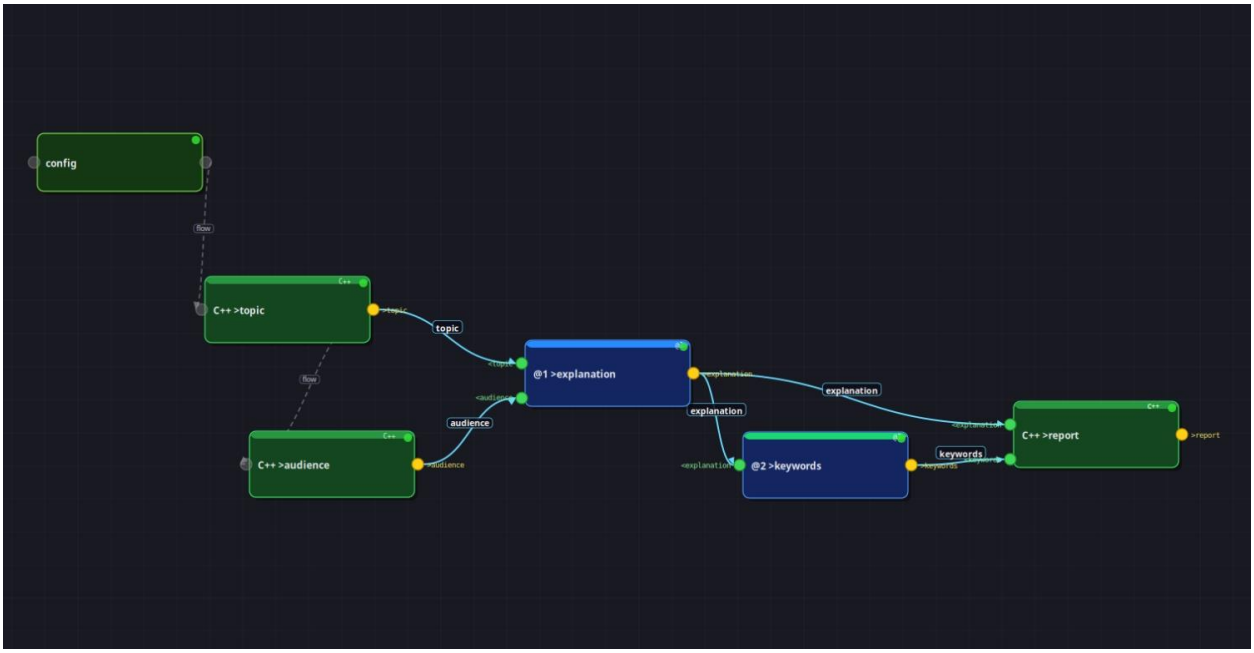
```
topic=smart pointers in C++
audience=junior C++ developer
max_sentences=3

cpp >topic #include <iostream> int main() {     std::cout << "smart pointers
in C++\n";     return 0; }

cpp >audience #include <iostream> int main() {     std::cout << "junior C++
developer\n";     return 0; }
```

```cpp <explanation report #include  #include  #include  std::string slurp(const char* key) { std::ifstream f(std::getenv(key)); return {std::istreambuf_iterator(f), {}}; } int main() { auto topic = slurp("MSH_VAR_topic"); auto audience = slurp("MSH_VAR_audience"); auto explanation = slurp("MSH_VAR_explanation"); auto keywords = slurp("MSH_VAR_keywords");

// trim trailing newlines for inline display
auto trim = [](std::string s) {
    while (!s.empty() && s.back() == '\n') s.pop_back();
    return s;
};

std::string sep(60, '=');
std::cout << sep << "\n"
          << "TOPIC    : " << trim(topic)    << "\n"
          << "AUDIENCE : " << trim(audience) << "\n"
          << sep << "\n"
          << "EXPLANATION:\n" << explanation
          << "KEYWORDS : "   << trim(keywords) << "\n"
          << sep << "\n";
return 0;

}


---

## Key Patterns
```

- The `config` block is **read-only documentation** — never rely on it to inj
ect runtime values.
- Each config parameter needs a matching `cpp >varname` block that `std::cout
`s the value.
- To create a reusable template, change only the C++ `std::cout` strings — th
e rest of the
  pipeline is unchanged.



Pattern 18 — FOREACH + Async LLM: Parallel Batch Processing

Concept: A C++ block creates a newline-separated list of items. FOREACH i
terates
line by line. On each iteration two async LLM calls run simultaneously — one
generates an
explanation, the other a real-world analogy. An `await=` barrier synchronises
them before
a C++ block prints the combined result.

Use case: Automated C++ algorithm cheat-sheet — for each algorithm name,
produce a
plain-English explanation and a real-world analogy in parallel, then display
them together.

Key concept:
- `async` inside FOREACH: for each item both LLMs run simultaneously — total
time per
  item = time of the slowest model.
- Each async call must have exactly one `>outvar` — otherwise `await=` cannot

match the job.
- The `await=` barrier lists all async output variables from the current iteration, comma-separated.

Flow Diagram

cpp >algorithms (“QuickSortSearch’s Algorithm”) ↓ [FOREACH algo in algorithms] @1 explanation async ─┐ @2 analogy async ─┤ cpp await=explanation,analogy ┘ cpp <explanation <analogy (print pair) [END_FOREACH] ↓ cpp (completion message)

Code

```cpp >algorithms
#include <iostream>
int main() {
    // one algorithm per line — mandatory for FOREACH line-by-line iteration
    std::cout << "QuickSort\nBinary Search\nDijkstra's Algorithm\n";
    return 0;
}
```

cpp await=explanation,analogy #include <iostream> int main() { return 0; }

```cpp <explanation <analogy #include  #include  #include  std::string slurp(const char* key) { std::ifstream f(std::getenv(key)); return {std::istreambuf_iterator(f), {}}; } int main() { std::ifstream fa(std::getenv(“MSH_VAR_algo”)); std::string algo((std::istreambuf_iterator(fa)), {}); while (!algo.empty() && algo.back() == ‘’) algo.pop_back();

std::cout << "=== " << algo << " ===\n"
          << "Explanation : " << slurp("MSH_VAR_explanation")
          << "Analogy     : " << slurp("MSH_VAR_analogy") << "\n";
return 0;

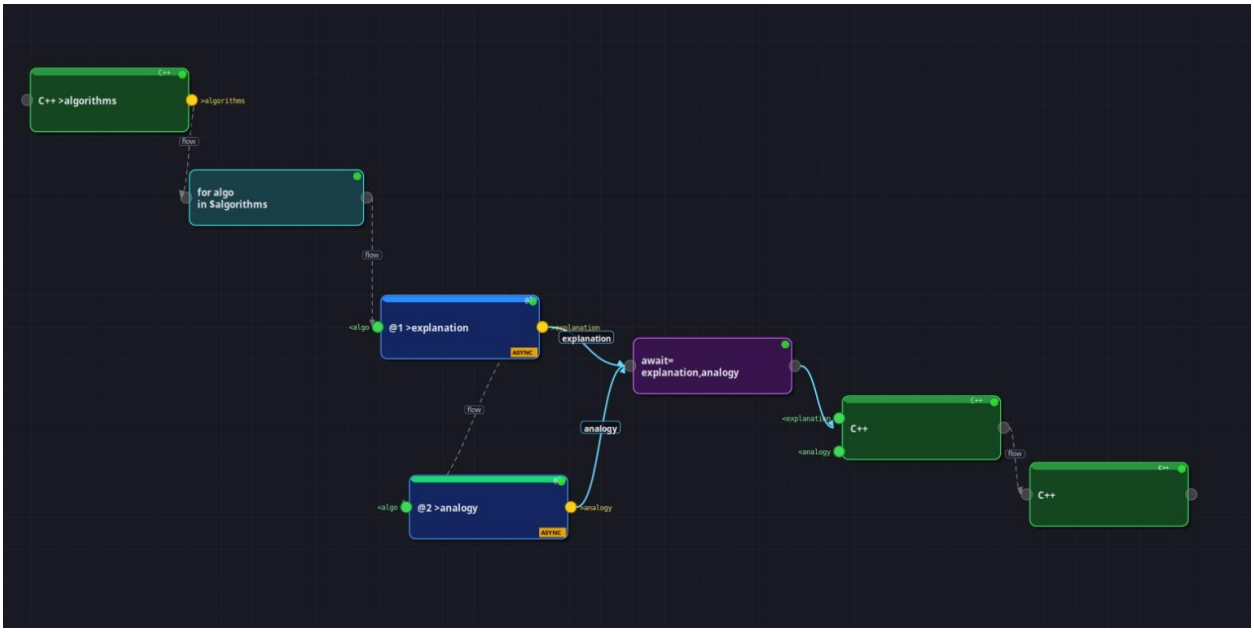
}


<!--@end_foreach-->

```cpp
#include <iostream>
int main() {
    std::cout << "=== Algorithm Cheat-Sheet complete ===\n";
    return 0;
}
```
```

Key Patterns

- `async` jobs are scoped to the current FOREACH iteration — `await=` only needs to list the variables written in that iteration.
- The `await=` C++ block can have an empty `main()` — its only purpose is to serve as the synchronisation barrier.
- Output variables (`explanation`, `analogy`) are **overwritten** on each iteration, so the print block must run before the next iteration begins (which it does, because `await=` guarantees it).



Pattern 19 — WHILE Quality Gate: Generate Until Threshold

Concept: Initialises a task, a `running/done` status flag, an iteration counter, a score, and an empty result. The WHILE loop runs while `status == running`. Each iteration: a C++ block increments the counter (writing directly to `$MSH_VAR_*`), LLM @1 generates a C++ code snippet, LLM @2 rates it 1–10 returning only the integer, a C++ block reads the score, prints a log line, and writes `done` to status when `score >= 8`.

Use case: Automated code quality gate — keep regenerating a C++ snippet until a scorer model judges it good enough.

Key concept: - Scorer (@2) must return a bare integer — prompt with "Reply with ONLY the integer." - Strip whitespace from the score string before numeric comparison in C++. -

The status-update C++ block must always write to `>status` — even when continuing. - All variables must be initialised before the loop (WHILE checks condition first).

Flow Diagram

```
cpp >task  >status="running"  >iteration="0"  >score="0"  >snippet=""
  ↓
[WHILE status:running]
  cpp <iteration >iteration          (counter++ via MSH_VAR_*)
  @1 <task >snippet                  (generate C++ snippet)
  @2 <snippet >score                 (rate 1-10, integer only)
  cpp <iteration <score <snippet >status   (log + threshold check)
[END_WHILE]
  ↓
cpp <snippet <score                  (print accepted snippet)
```

Code

```
cpp >task #include <iostream> int main() {     std::cout << "Write a modern
C++17 function template that safely clamps a value "
"between a minimum and maximum, using std::clamp or equivalent. "
"Must be generic, constexpr, and include a usage example in main().";
return 0; }
```

```
cpp >status #include <iostream> int main() { std::cout << "running\n"; return
0; }
```

```
cpp >iteration #include <iostream> int main() { std::cout << "0\n"; return 0;
}
```

```
cpp >score #include <iostream> int main() { std::cout << "0\n"; return 0; }
```

```
cpp >snippet #include <iostream> int main() { std::cout << "\n"; return 0; }
```

```
cpp <iteration >iteration #include <iostream> #include <fstream> int main() {
std::ifstream f(std::getenv("MSH_VAR_iteration"));     int val; f >> val;
f.close();     ++val;     std::ofstream w(std::getenv("MSH_VAR_iteration"));
w << val << "\n"; w.close();     // stdout must be only the number — mshell
captures it as the new variable value     std::cout << val << "\n";
return 0; }
```

```cpp <iteration <score status #include  #include  #include  #include  #include  int main() {
auto slurp = H:\const char* key -> std::string { std::ifstream f(std::getenv(key)); return
{std::istreambuf_iterator(f), {}}; }; std::string iter_s = slurp("MSH_VAR_iteration");
std::string score_s = slurp("MSH_VAR_score");

// strip whitespace from score
score_s.erase(std::remove_if(score_s.begin(), score_s.end(),
              [](char c){ return std::isspace((unsigned char)c); }),
```

```
            score_s.end());
int score = 0;
try { score = std::stoi(score_s); } catch(...) { score = 0; }

int iter = 0;
try { iter = std::stoi(iter_s); } catch(...) {}

std::cout << "[Iter " << iter << "] Score=" << score << "\n";

std::string new_status = (score >= 8) ? "done" : "running";
std::ofstream ws(std::getenv("MSH_VAR_status"));
ws << new_status << "\n"; ws.close();
std::cout << "[status -> " << new_status << "]\n";
return 0;

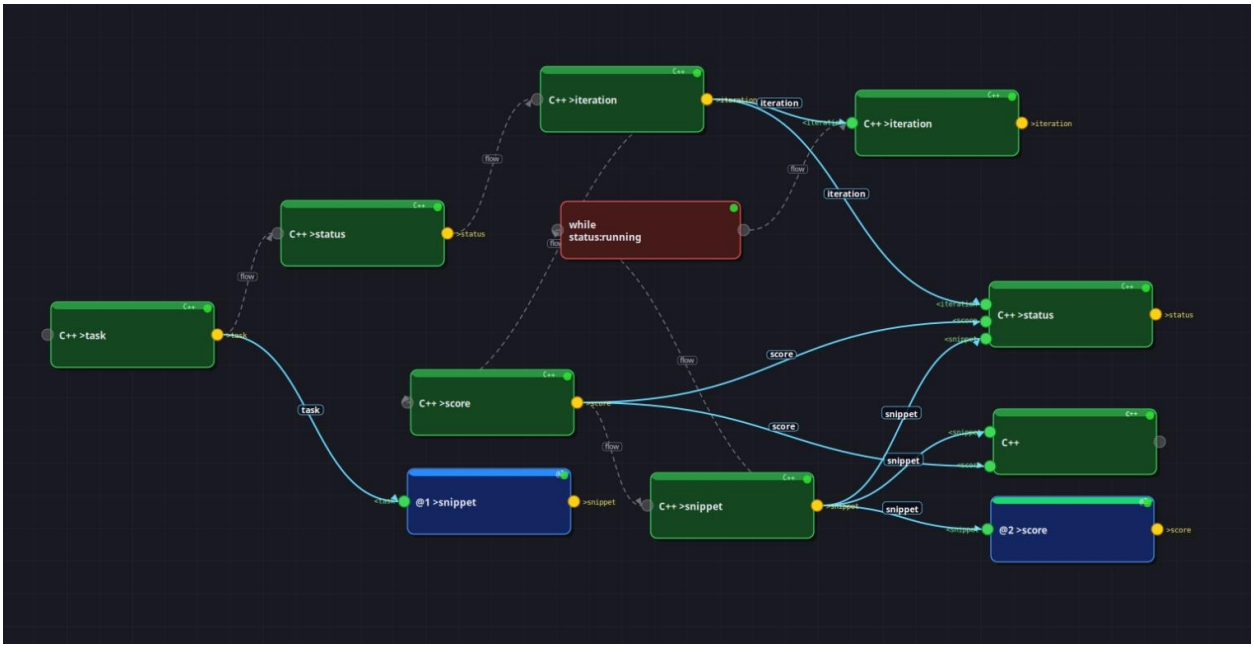
}


<!--@end_while-->

```cpp <snippet <score
#include <iostream>
#include <fstream>
#include <string>
int main() {
    std::ifstream fs(std::getenv("MSH_VAR_score"));
    std::string score_s((std::istreambuf_iterator<char>(fs)), {});
    std::ifstream fsnip(std::getenv("MSH_VAR_snippet"));
    std::string snippet((std::istreambuf_iterator<char>(fsnip)), {});
    std::cout << "=== Accepted Snippet (score=" << score_s << ") ===\n"
              << snippet << "\n";
    return 0;
}
```

Key Patterns

- Write to `>status` using `std::ofstream` inside the same C++ block that checks the threshold — both `>score` and `>status` are written in that one block.
- `std::remove_if` + `std::isspace` is the C++ equivalent of `tr -d '[:space:]'`.
- Always initialise all loop variables before the `<!--@while-->` comment.
- The status-write C++ block must always produce a value for `>status`, even when continuing — never leave it unwritten.
```



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

Concept: A C++ block stores a multi-sentence text. Three further C++ blocks split the text into chunks (sent1, sent2, sent3). Three async LLM @1 calls process each chunk in parallel — extracting the main concept in three words. An `await=` barrier waits for all three. The MERGE node marks the reduce point. LLM @2 synthesises the three labels into one coherent theme. A final C++ block prints the map results and the reduced result.

Use case: Parallel concept extraction from a long C++ design document — split the document into sections, extract the core idea from each section simultaneously, then synthesise an overall architectural theme.

Key concept: - Map phase: N parallel LLMs, each on its own chunk — total time = slowest chunk. - Reduce phase: one LLM receives all results via multiple `<invar`; mshell injects `[varname]:` labels for each. - AWAIT before MERGE is mandatory — MERGE is a visual marker only.

Flow Diagram

```
cpp >raw_text
  ↓
cpp <raw_text >sent1 / cpp <raw_text >sent2 / cpp <raw_text >sent3  (split)
  @1 <sent1 >analysis1  async ─┐
  @1 <sent2 >analysis2  async ─┤  MAP
  @1 <sent3 >analysis3  async ─┤
  cpp await=analysis1,analysis2,analysis3 ─┘
```

```
[MERGE]
  @2 <analysis1 <analysis2 <analysis3 >summary   (REDUCE)
  ↓
cpp <analysis1 <analysis2 <analysis3 <summary    (print)
```

Code

```
cpp >raw_text #include <iostream> int main() {     std::cout <<         "C++
templates enable compile-time polymorphism and zero-cost abstractions. "
"The RAII idiom ties resource lifetimes to object scopes, eliminating leaks.
"         "Move semantics transfer ownership of resources without copying,
maximising throughput. "         "Concepts constrain template parameters,
providing clear error messages. "         "Coroutines allow cooperative
multitasking with minimal overhead."         << std::endl;     return 0; }
```

```
cpp <raw_text >sent1 #include <iostream> #include <fstream> #include <string>
#include <sstream> #include <vector> int main() {     std::ifstream
f(std::getenv("MSH_VAR_raw_text"));     std::string
text((std::istreambuf_iterator<char>(f)), {});     std::vector<std::string>
sents;     std::string s;     for (char c : text) {         s += c;
if (c == '.') { sents.push_back(s); s.clear(); }     }     if (sents.size()
>= 1) std::cout << sents[0] << "\n";     return 0; }
```

```
cpp <raw_text >sent2 #include <iostream> #include <fstream> #include <string>
#include <vector> int main() {     std::ifstream
f(std::getenv("MSH_VAR_raw_text"));     std::string
text((std::istreambuf_iterator<char>(f)), {});     std::vector<std::string>
sents;     std::string s;     for (char c : text) {         s += c;
if (c == '.') { sents.push_back(s); s.clear(); }     }     // sentences 1 and
2 (0-indexed)     std::string out;     if (sents.size() >= 2) out +=
sents[1];     if (sents.size() >= 3) out += " " + sents[2];     std::cout <<
out << "\n";     return 0; }
```

```
cpp <raw_text >sent3 #include <iostream> #include <fstream> #include <string>
#include <vector> int main() {     std::ifstream
f(std::getenv("MSH_VAR_raw_text"));     std::string
text((std::istreambuf_iterator<char>(f)), {});     std::vector<std::string>
sents;     std::string s;     for (char c : text) {         s += c;
if (c == '.') { sents.push_back(s); s.clear(); }     }     std::string out;
if (sents.size() >= 4) out += sents[3];     if (sents.size() >= 5) out += " "
+ sents[4];     std::cout << out << "\n";     return 0; }
```

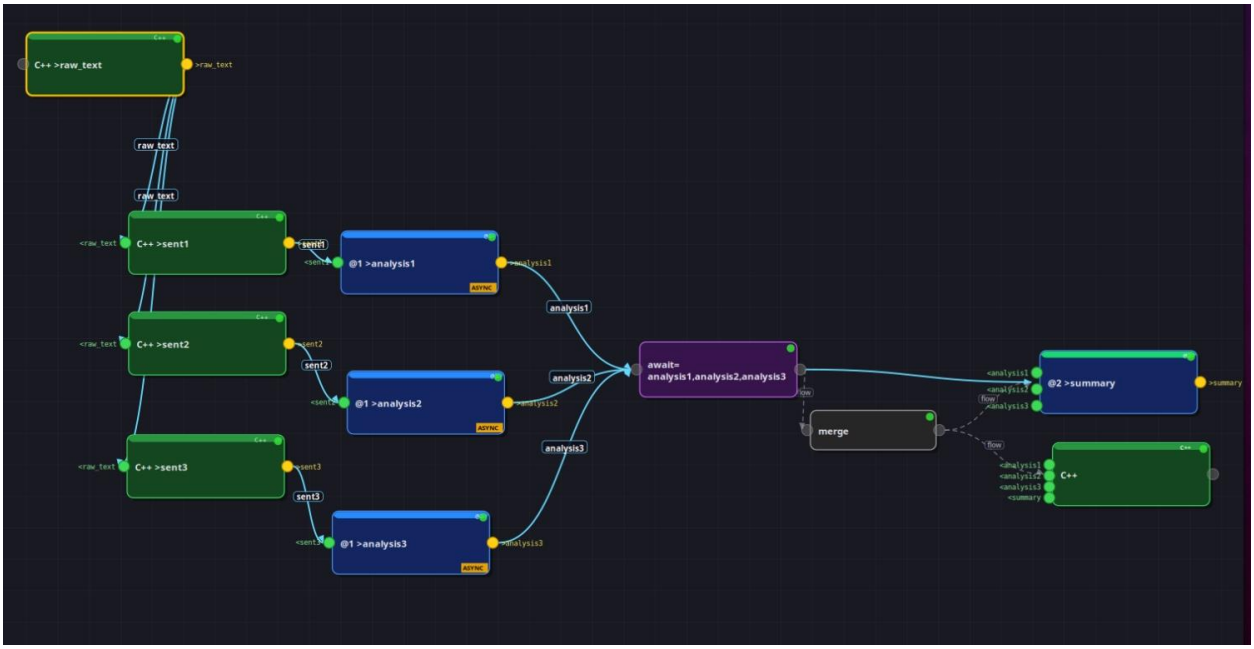
```
cpp await=analysis1,analysis2,analysis3 #include <iostream> int main() {
std::cout << "[await] all three map jobs complete\n";     return 0; }
```

```
cpp <analysis1 <analysis2 <analysis3 <summary #include <iostream> #include
<fstream> #include <string> std::string slurp(const char* key) {
std::ifstream f(std::getenv(key));     return
{std::istreambuf_iterator<char>(f), {}}; } int main() {     std::cout << "===
Map Results ===\n"               << "Chunk 1: " << slurp("MSH_VAR_analysis1")
```

```
<< "Chunk 2: " << slurp("MSH_VAR_analysis2")              << "Chunk 3: " <<
slurp("MSH_VAR_analysis3")              << "\n=== Reduce / Theme ===\n"
<< slurp("MSH_VAR_summary") << "\n";     return 0; }
```

Key Patterns

- Three independent C++ blocks each parse `raw_text` to extract different sentence ranges — clean, compiled, no shell scripting required.
- `await=analysis1,analysis2,analysis3` must list **all** async output variables before MERGE.
- The reduction LLM (@2) receives `[analysis1]:`, `[analysis2]:`, `[analysis3]:` labels automatically when multiple `<invar` are provided.



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

Concept: Initialises a task, `result=fail`, and `last_error=none`. A LOOP runs up to 3 times, exiting early when `result == ok`. Each iteration: LLM @1 generates C++ code (seeing the previous error or "none"). A C++ block prints the generated code. The TRY block compiles and runs it — writing "ok" on success. CATCH writes "fail" to result without reading the catch variable. On the next iteration LLM @1 sees `last_error` and self-corrects.

Use case: Automated self-healing C++ code generation — if the generated code fails to compile or crashes, retry with error context until it works.

Key concept: - CATCH block does **not** use `<last_error` — writes `fail` to `>result` directly. - `last_error` is initialised to `"none"` before the loop. - `result` is initialised to `"fail"` before the loop. - LOOP reads the last non-empty line of `result` for the `until=` check.

Flow Diagram

```
cpp >task  >result="fail"  >last_error="none"
  ↓
[LOOP max=3 until=result:ok]
  @1 <task <last_error >code      (generate / fix C++ code)
  cpp <code                       (print generated code)
  [TRY]
    cpp <code >result             (compile + run; writes "ok" on success)
  [CATCH >last_error]
    cpp >result                   (writes "fail")
  [END_TRY]
[END_LOOP]
  ↓
cpp <result                       (final status)
```

Code

```
cpp >task #include <iostream> int main() {     std::cout <<         "Write a C++ program that reads a JSON-like string "         "'{\"name\": \"Alice\", \"age\": 30}' hardcoded as a std::string, "         "manually parses the 'name' value (no external libraries), "         "and prints it. Must compile with g++ -std=c++17 -O2."         << std::endl;     return 0; }
```

```
cpp >result #include <iostream> int main() { std::cout << "fail\n"; return 0; }
```

```
cpp >last_error #include <iostream> int main() { std::cout << "none\n"; return 0; }
```

```
cpp <code #include <iostream> #include <fstream> #include <string> int main() {     std::cout << "=== Generated Code ===\n";     std::ifstream f(std::getenv("MSH_VAR_code"));     std::string line;     while (std::getline(f, line)) std::cout << line << "\n";     return 0; }
```

```
cpp <code #include <iostream> #include <fstream> #include <string> #include <cstdlib> int main() {     const char* code_path = std::getenv("MSH_VAR_code");     // copy to stable .cpp path before compiling std::string cp_cmd = std::string("cp -- \"") + code_path + "\" /tmp/msh_p21_gen.cpp";     std::system(cp_cmd.c_str());     // compile     if (std::system(             "g++ -std=c++17 -O2 -o /tmp/msh_p21_bin /tmp/msh_p21_gen.cpp 2>/tmp/msh_p21_err.txt"         ) != 0) {         std::ifstream ef("/tmp/msh_p21_err.txt");         std::string err((std::istreambuf_iterator<char>(ef)), {});         std::cerr << err;
```

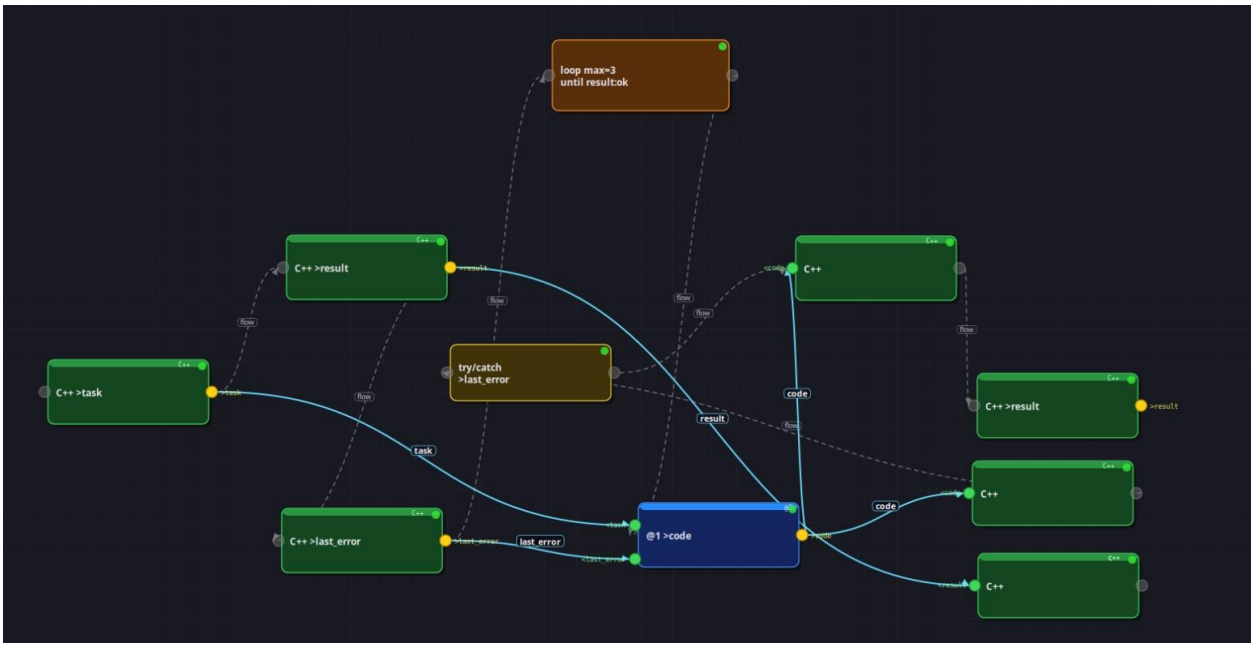
```
std::exit(1);     }     // run — stdout goes to terminal only (no >outvar on this block)     std::system("/tmp/msh_p21_bin");     // write ok to result file directly     std::ofstream wr(std::getenv("MSH_VAR_result"));     wr << "ok\n"; wr.close();     // print ok as last stdout line — LOOP checks last stdout line     std::cout << "ok\n";     return 0; }
```

```
cpp >result #include <iostream> #include <fstream> #include <string> int main() {     std::cout << "=== Error on this attempt: try_block_failed ===\n";     // write fail to result     std::ofstream wr(std::getenv("MSH_VAR_result"));     wr << "fail\n"; wr.close(); std::cout << "fail\n";     return 0; }
```

```
cpp <result #include <iostream> #include <fstream> #include <string> int main() {     std::ifstream f(std::getenv("MSH_VAR_result"));     std::string r((std::istreambuf_iterator<char>(f)), {});     std::cout << "=== Final status: " << r << " ===\n";     return 0; }
```

Key Patterns

- TRY fires CATCH when the C++ block exits with a non-zero code (`std::exit(1)`).
- CATCH declares `>last_error` so the parser registers it, but the block does **not** read it — write `fail` to `>result` directly with `std::ofstream`.
- The compile error is sent to stderr so mshell logs it, but the CATCH mechanism is triggered solely by the non-zero exit code.
- `LOOP reads the last non-empty line` — always write ok or `fail` as the only line.



Pattern 22 — Multi-Variable Output: Structured Field Extraction

Concept: A C++ block stores a technical description as input. LLM @1 responds in a strict three-line format. A C++ block with three output variables parses the response with `std::regex` and writes each field directly to the corresponding `$MSH_VAR_*` file. LLM @2 rewrites the summary for the detected audience level.

Use case: Structured metadata extraction from C++ library documentation — split a raw description into summary, key types, and audience level; then adapt the summary for that audience.

Key concept: - Multiple `>outvar` on a C++ block: mshell runs the block without capturing stdout — the block MUST write to `$MSH_VAR_*` files directly via `std::ofstream`. - Multiple `>outvar` on an LLM directive: the FULL response is copied identically into each variable — it does NOT split the response. - `MSH_VAR_*` environment variables are pre-set by the parser before running the block.

Flow Diagram

```
cpp >input
  ↓
@1 <input >raw_response             (LLM: strict 3-line format)
cpp <raw_response >summary >types >audience
  (parse with std::regex; write fields via std::ofstream to MSH_VAR_*)
@2 <summary <audience >adaptation   (LLM: rewrite for audience)
cpp <adaptation                     (print)
```

Code

```
cpp >input #include <iostream> int main() {     std::cout <<
"std::unordered_map is a hash-table-based associative container introduced "
"in C++11 that provides average O(1) lookup, insertion, and deletion. "
"It stores key-value pairs with no guaranteed ordering, making it ideal for "
"fast lookups when iteration order does not matter."         << std::endl;
return 0; }
```

```cpp summary >types >audience #include  #include  #include  #include  int main() { // read raw_response std::ifstream fr(std::getenv("MSH_VAR_raw_response")); std::string text((std::istreambuf_iterator(fr)), {});

```
// parse with regex
auto extract = [&](const std::string& label) -> std::string {
    std::regex re(label + ":\\s*(.+)");
    std::smatch m;
    if (std::regex_search(text, m, re)) return m[1].str();
    return "n/a";
};
```
```

```
std::string summary  = extract("SUMMARY");
std::string types    = extract("TYPES");
std::string audience = extract("AUDIENCE");

// write each field to its variable file directly
{
    std::ofstream ws(std::getenv("MSH_VAR_summary"));
    ws << summary << "\n";
}
{
    std::ofstream wt(std::getenv("MSH_VAR_types"));
    wt << types << "\n";
}
{
    std::ofstream wa(std::getenv("MSH_VAR_audience"));
    wa << audience << "\n";
}

// stdout is NOT captured (multiple outvars) — goes to terminal log only
std::cout << "Summary  : " << summary  << "\n"
          << "Types    : " << types    << "\n"
          << "Audience : " << audience << "\n";
return 0;

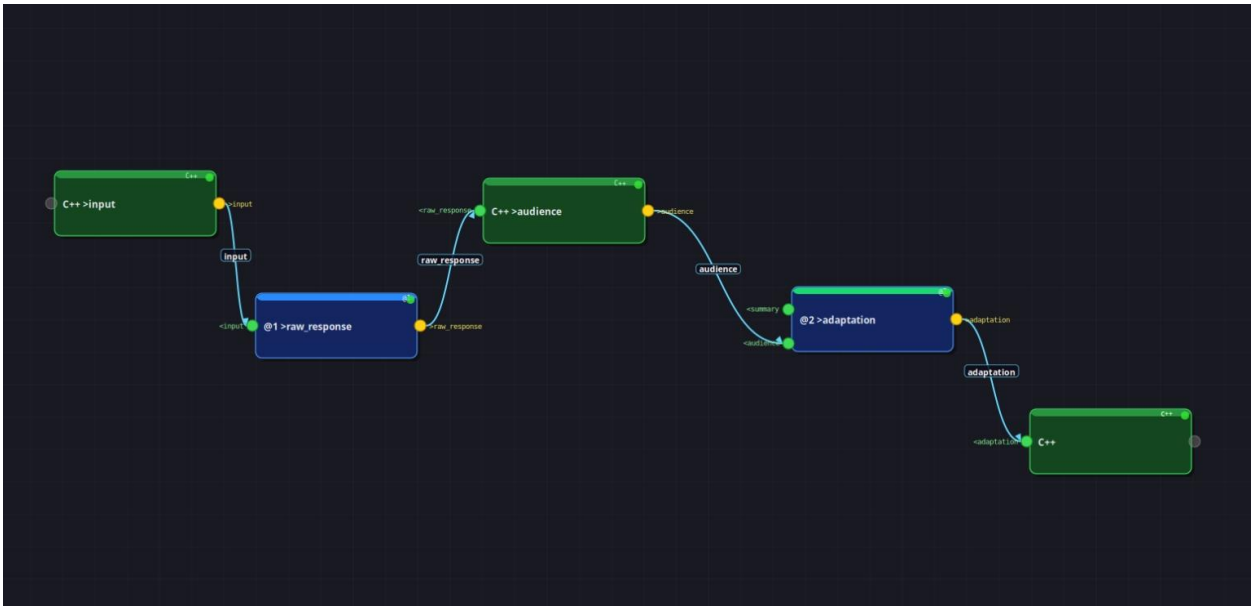
}


<!--@2 <summary <audience >adaptation
The first input is a technical summary. The second input is the target audien
ce level
(beginner / intermediate / expert).
Rewrite the summary for that audience in one sentence.
Use vocabulary and analogies appropriate for that level.
-->

```cpp <adaptation
#include <iostream>
#include <fstream>
#include <string>
int main() {
    std::ifstream f(std::getenv("MSH_VAR_adaptation"));
    std::string a((std::istreambuf_iterator<char>(f)), {});
    std::cout << "\n=== Adapted Version ===\n" << a << "\n";
    return 0;
}
```
```

Key Patterns

- **Multiple `>outvar` on a C++ block:** always write directly to `std::getenv("MSH_VAR_*")` with `std::ofstream` — stdout is ignored by the parser in this mode.
- `std::regex` is the idiomatic C++11 way to parse structured LLM output.
- Each `std::ofstream` write should be scoped (use `{}` blocks) so the file is flushed and closed before the next write.
- Validate extracted fields and fall back to `"n/a"` to avoid empty variable files.



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

Concept: A CONFIG block documents the parameters. C++ blocks initialise all runtime variables. The WHILE loop runs while `status == running`. Each iteration: a C++ block increments the counter, LLM @1 generates a C++ code explanation, LLM @2 scores it for clarity and correctness (1–10), a C++ block checks the score and sets `done` when `quality >= 7`. After the loop LLM @3 polishes the final explanation for publication. A C++ block prints the result with metrics.

Use case: Fully parameterised C++ teaching pipeline — automatically iterate until the explanation of a C++ concept meets a quality bar, then polish it for documentation. Change two C++ `std::cout` lines to switch topic and audience.

Key concept: - Three distinct model roles: generator (@1), scorer (@2), finisher (@3). - CONFIG makes the pipeline a reusable template. - @2 receives `<target_audience` explicitly — the scorer needs audience context.

Flow Diagram

```
config (subject, target_audience, quality_threshold — doc only)
cpp >subject  >target_audience  >status="running"  >iteration="0"
cpp >quality="0"  >explanation=""
  ↓
[WHILE status:running]
  cpp <iteration >iteration
  @1 <subject <target_audience >explanation   (generate)
  @2 <explanation <target_audience >quality   (score 1-10)
  cpp <iteration <quality >status             (log + threshold)
[END_WHILE]
  ↓
@3 <explanation >final_polish                 (polish)
cpp <final_polish <quality <iteration         (print with metrics)
```

Code

```
subject=std::shared_ptr and ownership semantics
target_audience=intermediate C++ developer
quality_threshold=7

cpp >subject #include <iostream> int main() {     std::cout <<
"std::shared_ptr and ownership semantics\n";     return 0; }

cpp >target_audience #include <iostream> int main() {     std::cout <<
"intermediate C++ developer\n";     return 0; }

cpp >status #include <iostream> int main() { std::cout << "running\n"; return
0; }

cpp >iteration #include <iostream> int main() { std::cout << "0\n"; return 0;
}

cpp >quality #include <iostream> int main() { std::cout << "0\n"; return 0; }

cpp >explanation #include <iostream> int main() { std::cout << "\n"; return
0; }

cpp <iteration >iteration #include <iostream> #include <fstream> int main() {
std::ifstream f(std::getenv("MSH_VAR_iteration"));     int val; f >> val;
f.close();     ++val;     std::ofstream w(std::getenv("MSH_VAR_iteration"));
w << val << "\n"; w.close();     std::cout << val << "\n";     return 0; }
```

```cpp <iteration status #include  #include  #include  #include  int main() { auto slurp = H:\const char* key -> std::string { std::ifstream f(std::getenv(key)); return {std::istreambuf_iterator(f), {}}; }; std::string iter_s = slurp("MSH_VAR_iteration"); std::string score_s = slurp("MSH_VAR_quality");

// strip whitespace
score_s.erase(std::remove_if(score_s.begin(), score_s.end(),
              [](char c){ return std::isspace((unsigned char)c); }),
```

```
              score_s.end());
int score = 0;
try { score = std::stoi(score_s); } catch(...) {}

std::cout << "[Iter " << iter_s << "] Quality: " << score << "\n";

std::string new_status = (score >= 7) ? "done" : "running";
std::ofstream ws(std::getenv("MSH_VAR_status"));
ws << new_status << "\n"; ws.close();
return 0;

}

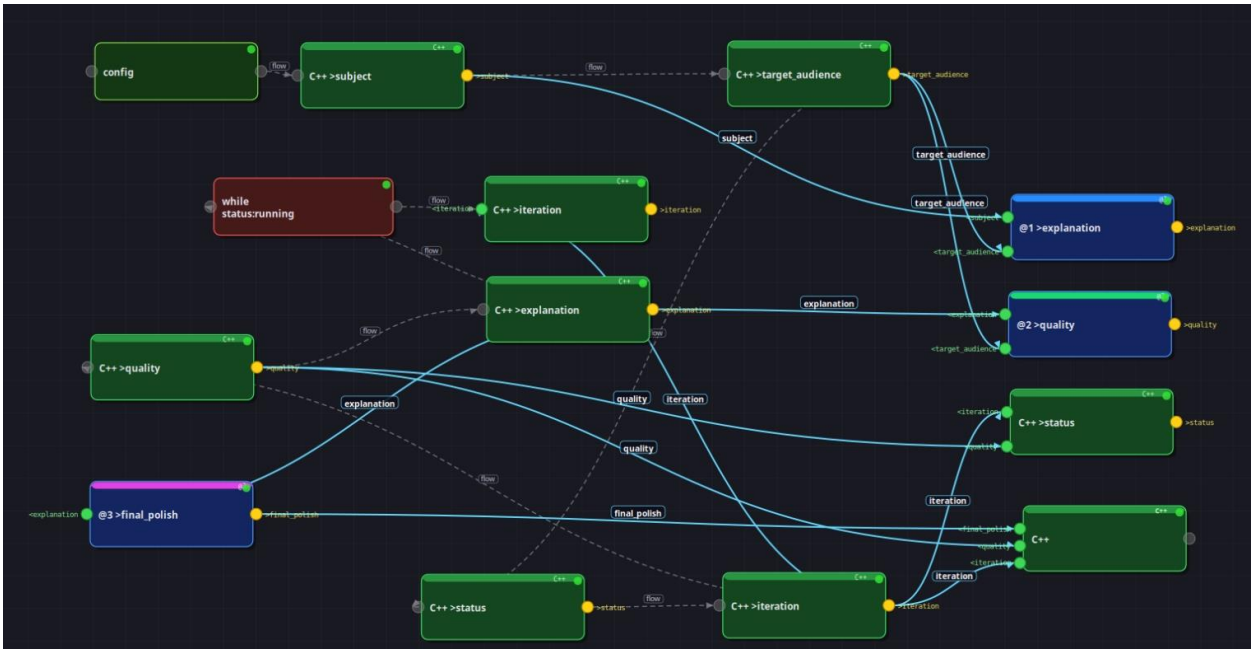

<!--@end_while-->

<!--@3 <explanation >final_polish
The input is a C++ technical explanation that has passed a quality review.
Polish it lightly for final publication in developer documentation.
Keep exactly 3 sentences. No markdown formatting. No code fences.
-->

```cpp <final_polish <quality <iteration
#include <iostream>
#include <fstream>
#include <string>
std::string slurp(const char* key) {
    std::ifstream f(std::getenv(key));
    return {std::istreambuf_iterator<char>(f), {}};
}
int main() {
    auto polish  = slurp("MSH_VAR_final_polish");
    auto quality = slurp("MSH_VAR_quality");
    auto iter    = slurp("MSH_VAR_iteration");
    // trim
    auto trim = [](std::string s) {
        while (!s.empty() && s.back() == '\n') s.pop_back();
        return s;
    };
    std::string sep(60, '=');
    std::cout << sep << "\n"
              << "Final (score=" << trim(quality)
              << ", iterations=" << trim(iter) << ")\n"
              << sep << "\n"
              << polish
              << sep << "\n";
    return 0;
}
```
```

Key Patterns

- CONFIG parameters each need a matching `cpp >varname` block — CONFIG itself injects nothing.
- @2 receives `<target_audience` **explicitly** so the scorer knows who it's scoring for.
- Three model roles: generator (@1), scorer (@2), finisher (@3) — each has a single focused responsibility.
- Strip whitespace from the score string in C++ before numeric comparison.



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

Concept: A C++ block creates a newline-separated list of C++ expressions (some valid, one intentionally broken). FOREACH iterates line by line. For each item the TRY block runs a C++ block that compiles and evaluates the expression. On success LLM @1 describes the result; a C++ block prints `[OK]`. On failure (non-zero exit) CATCH prints `[ERR]` as a hardcoded literal. The loop continues to the next item regardless.

Use case: Batch C++ expression evaluator with per-expression error isolation — one malformed expression never stops the entire batch.

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

Flow Diagram

```
cpp >expressions        (newline-separated list; one broken)
  ↓
[FOREACH expr in expressions]
  [TRY]
    cpp <expr >result   (compile + evaluate — may exit(1))
    @1 <result >insight (LLM: describe the result)
    cpp <insight        (print [OK])
  [CATCH >expr_error]
    cpp                 (print [ERR] literal string)
  [END_TRY]
[END_FOREACH]
  ↓
cpp                     (completion message)
```

Code

```
cpp >expressions #include <iostream> int main() {     // Three expressions,
one line each; the second is intentionally invalid C++     std::cout << "1 +
2 * 3\n"               << "int x = @@@INVALID@@@\n"               <<
"std::string(5, 'X')\n";     return 0; }
```

```cpp result #include  #include  #include  #include  int main() { std::ifstream f(std::getenv("MSH_VAR_expr")); std::string expr((std::istreambuf_iterator(f)), {}); // trim while (!expr.empty() && (expr.back() == '' || expr.back() == ' ')) expr.pop_back();

// Write a tiny C++ program that evaluates the expression and prints it
std::string src =
    "#include <iostream>\n"
    "#include <string>\n"
    "int main() {\n"
    "    auto val = (" + expr + ");\n"
    "    std::cout << val << \"\\n\";\n"
    "    return 0;\n"
    "}\n";

// write to temp file
{
    std::ofstream tsrc("/tmp/msh_p24_expr.cpp");
    tsrc << src;
}

// compile
int ret = std::system(
    "g++ -std=c++17 -O2 -o /tmp/msh_p24_expr_bin /tmp/msh_p24_expr.cpp 2>/tmp
/msh_p24_err.txt");
if (ret != 0) {
    std::ifstream ef("/tmp/msh_p24_err.txt");
```

```
    std::string err((std::istreambuf_iterator<char>(ef)), {});
    std::cerr << err;
    std::exit(1);
}

// run and capture output into result variable
std::system("/tmp/msh_p24_expr_bin");
return 0;

}


<!--@1 <result >insight
The input contains the evaluated result of a C++ expression.
In one sentence, describe what kind of value this is and what operation produ
ced it.
-->

```cpp <insight
#include <iostream>
#include <fstream>
#include <string>
int main() {
    std::ifstream fi(std::getenv("MSH_VAR_insight"));
    std::string insight((std::istreambuf_iterator<char>(fi)), {});
    std::cout << "[OK] " << insight << "\n";
    return 0;
}
```

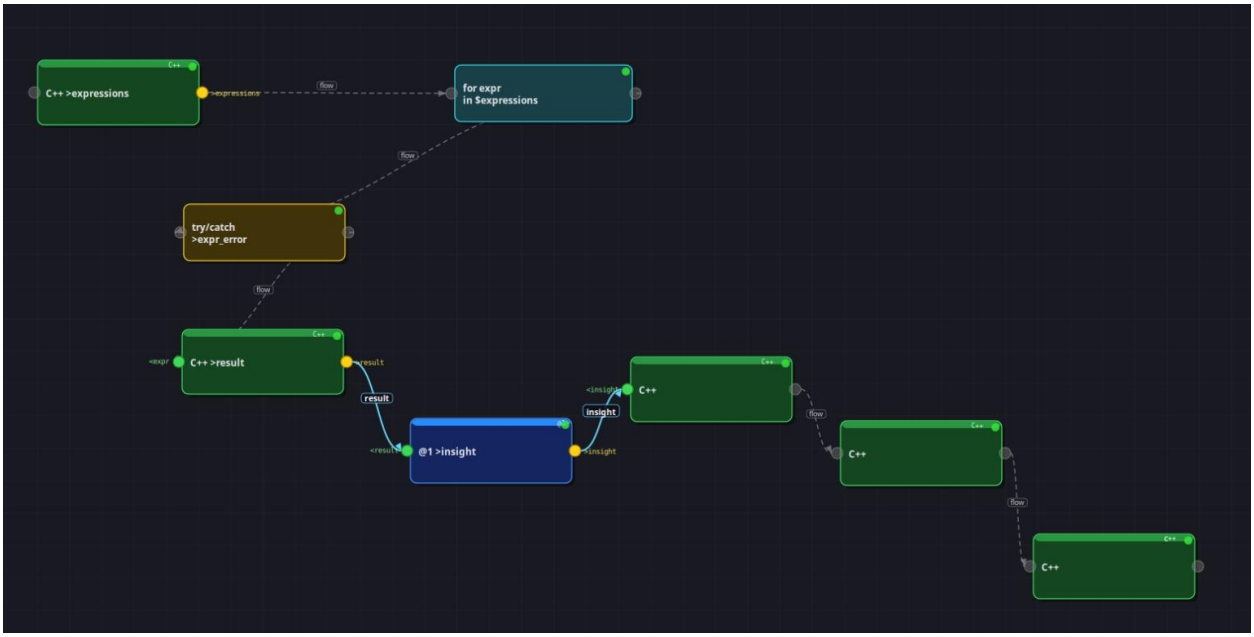
```cpp
#include <iostream>
int main() {
    std::cout << "[ERR] Expression failed: try_block_failed\n";
    return 0;
}
```

```cpp
#include <iostream>
int main() {
    std::cout << "=== Batch complete. Errors were isolated, pipeline never st
opped. ===\n";
    return 0;
}
```

Key Patterns

- TRY/CATCH inside FOREACH = **per-item error isolation** — a broken expression never halts the remaining iterations.
- The C++ TRY block dynamically generates, compiles, and runs a tiny program for each expression — all without leaving C++.
```

- CATCH block does **not** reference `<expr_error` — it prints the literal `"try_block_failed"` string.
- Do **not** reference `result` or `insight` outside the TRY block — if TRY fails they were never written.
- `std::exit(1)` in the compile-failure branch is the idiomatic way to trigger CATCH from C++.



Complete Summary Table

#	Pattern	Nodes	Models	Key Feature
1	Linear Pipeline	—	—	Sequential C++ stages, file-based data flow
2	LLM in Middle	—	@1	LLM as transformer between C++ stages
3	Fan-Out	—	@1	One variable, many independent C++ consumers
4	Code Gen + Exec	—	@1	cp to .cpp → g++ → run
5	Two-LLM Review	—	@1, @2	Generate → Review → Improve → Compile
6	Parallel 3 Models	—	@1–@3	Same question to 3 models, C++ collects
7	Eval-Optimizer Loop	LOOP	@1, @2	Loop until ACCEPTED; verdict prints first word
8	Multi-Stage + LLMs	—	@1, @2	Full C++ pipeline + narrative + headline

#	Pattern	Nodes	Models	Key Feature
9	Routing	—	@1	LLM classifies → conditional C++ branch
10	Full Pipeline	AWAIT	@1–@3	Sieve → stats → async poem + analysis → frame
11	MShell Node	mshell	@1, @2	Native mshell AI calls + C++ formatting
12	Async + Await	AWAIT	@1–@3	3 async LLMs → barrier → synthesis
13	WHILE Counter	WHILE	@1	File-based status exit; LLM fact per iteration
14	FOREACH List	FOREACH	@1	Line-by-line iteration over C++ feature list
15	TRY/CATCH	TRY/CATCH	—	exit(1) triggers CATCH; no `<errvar` in body
16	SPLIT + MERGE	SPLIT, MERGE	@1, @2	Async parallel analysis + merge synthesis
17	CONFIG Pipeline	CONFIG	@1, @2	Parameterised template; CONFIG is doc-only
18	FOREACH + Async	FOREACH, AWAIT	@1, @2	Per-item parallel LLM calls
19	WHILE Quality Gate	WHILE	@1, @2	Score threshold; stdout = number only
20	Map-Reduce	SPLIT, MERGE, AWAIT	@1, @2	3-chunk async map + reduce synthesis
21	TRY/CATCH + LOOP	TRY/CATCH, LOOP	@1	Retry compile; no >result on TRY block
22	Multi-Var Output	—	@1, @2	std::regex field extraction → ofstream writes
23	CONFIG + WHILE + 3M	CONFIG, WHILE	@1–@3	Adaptive pipeline; 3 distinct model roles
24	FOREACH + TRY/CATCH	FOREACH, TRY/CATCH	@1	Per-item compile isolation; CATCH is literal

Verified Rules

Rule	Detail
cp before g++	Always `cp MSH_VAR_code /tmp/gen.cpp` before compiling. Passing the raw mshell context path to `g++` causes `file format not recognized` linker error.
LOOP	LOOP checks **last stdout line** of last block in body. Print the exit value

Rule	Detail
stdout	(`ACCEPTED`, `ok`) as the final `std::cout` line.
WHILE file	WHILE checks **MSH_VAR_status file** between iterations. Write via `std::ofstream`, no need to print to stdout.
>outvar stdout	When a block declares `>outvar`, mshell captures **all stdout** as the variable value. Avoid printing program output if you also need the var to equal `ok`/`fail`.
Counter stdout	A block with `>counter` must print **only the number** to stdout — any extra text becomes the variable value and breaks `std::stoi`.
CATCH body	Never use `<errvar` inside CATCH. Print `"try_block_failed"` or your own message directly.
CONFIG	CONFIG is documentation only — never injects runtime values. Always pair with `cpp >varname` blocks.
Multiple >outvar	Block must write directly to `std::getenv("MSH_VAR_*")` via `std::ofstream`. stdout is ignored.

Appendix I: Code examples for each pattern

Sent to mshell (4064 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 1 Linear Data Pipeline (Pure C++)

Concept: Sequential data flow where each C++ block reads the previous block's output and passes its own result forward. Demonstrates that a complete, multi-stage computation can be expressed in a single mshell document using only one language.

Use case: Multi-stage numerical pipelines - raw generation - statistics - formatting analysis - aggregation - reporting, each stage a standalone C++ binary.

Key concept: Variables flow strictly top-to-bottom. Each `>outvar` producer must appear before any `<outvar` consumer in the document.

Flow Diagram

```
cpp >raw_number
cpp <raw_number >squared_stats      (compute square, cube, sqrt)
cpp <squared_stats >formatted       (format as a report string)
cpp <formatted >enriched            (append prime-check result)
cpp <enriched >filtered             (uppercase all letters)
cpp <filtered >final_report         (wrap in decorative frame)
cpp <final_report                   (print to stdout)
```
Code
```cpp >raw_number
#include <iostream>
int main() {
    std::cout << 42 << std::endl;
    return 0;
}
```
```cpp <raw_number >squared_stats
#include <iostream>
#include <fstream>
#include <cmath>
#include <string>
int main() {
    std::ifstream f(std::getenv("MSH_VAR_raw_number"));
    int val; f >> val;
    long long sq = (long long)val * val;
    long long cu = sq * val;
```

```
   double sr = std::sqrt((double)val);
   std::cout << "value=" << val
             << " square=" << sq
             << " cube=" << cu
             << " sqrt=" << sr
             << std::endl;
   return 0;
}
```

```cpp <squared_stats >formatted
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
int main() {
   std::ifstream f(std::getenv("MSH_VAR_squared_stats"));
   std::string line; std::getline(f, line);
   std::cout << "[Stage 3 / Formatted] " << line << std::endl;
   return 0;
}
```

```cpp <formatted >enriched
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
// naive primality check extracted from the stats value
```

```
bool is_prime(int n) {
   if (n < 2) return false;
   for (int i = 2; i * i <= n; i++) if (n % i == 0) return false;
   return true;
}
int main() {
   std::ifstream f(std::getenv("MSH_VAR_formatted"));
   std::string line; std::getline(f, line);
   // extract the raw value from "value=N ..."
   int val = 0;
   std::istringstream ss(line);
   std::string tok;
   while (ss >> tok) {
      if (tok.rfind("value=", 0) == 0) val = std::stoi(tok.substr(6));
   }
   std::string prime_label = is_prime(val) ? "prime" : "composite";
   std::cout << line << " prime=" << prime_label << std::endl;
   return 0;
}
```

```cpp <enriched >filtered
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
int main() {
   std::ifstream f(std::getenv("MSH_VAR_enriched"));
```

```
    std::string line; std::getline(f, line);
    for (char& c : line) c = std::toupper(c);
    std::cout << line << std::endl;
    return 0;
}
```

```cpp <filtered >final_report
#include <iostream>
#include <fstream>
#include <string>
int main() {
    std::ifstream f(std::getenv("MSH_VAR_filtered"));
    std::string line; std::getline(f, line);
    std::string border(line.size() + 4, '=');
    std::cout << border << "\n"
              << "| " << line << " |\n"
              << border << std::endl;
    return 0;
}
```

```cpp <final_report
#include <iostream>
#include <fstream>
#include <string>
int main() {
    std::ifstream f(std::getenv("MSH_VAR_final_report"));
    std::string line;
```

```
    while (std::getline(f, line)) std::cout << line << "\n";
    return 0;
}
```

Key Patterns

- Every stage is a self-contained `g++`-compiled binary; mshell compiles and runs each block automatically.

- `std::getenv("MSH_VAR_varname")` returns the **file path** of the variable \u2014 open it with `std::ifstream`.

- No shared memory, no global state \u2014 pure file-based data flow.

42

value=42 square=1764 cube=74088 sqrt=6.48074

[Stage 3 / Formatted] value=42 square=1764 cube=74088 sqrt=6.48074

[Stage 3 / Formatted] value=42 square=1764 cube=74088 sqrt=6.48074 prime=composite

[STAGE 3 / FORMATTED] VALUE=42 SQUARE=1764 CUBE=74088 SQRT=6.48074 PRIME=COMPOSITE

==========

| [STAGE 3 / FORMATTED] VALUE=42 SQUARE=1764 CUBE=74088 SQRT=6.48074 PRIME=COMPOSITE |

==========

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

| [STAGE 3 / FORMATTED] VALUE=42 SQUARE=1764 CUBE=74088 SQRT=6.48074 PRIME=COMPOSITE |

==========

/home/igor > Sent to mshell (2900 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 2 LLM in the Middle

Concept: A C++ block generates structured data, an LLM transforms it with natural language intelligence, and a second C++ block processes the model's response for further use.
Use case: Automated log analysis C++ generates a synthetic server-error log, the LLM produces a human-readable incident summary, and C++ extracts word/sentence statistics from the summary.

Key concept: LLM directive `<!--@1 <data >result ... -->` injects the variable file contents into the prompt and writes the model response to `>result`, which any subsequent block can consume exactly like any other variable.
Flow Diagram
```
cpp >log_data          (generate synthetic error log)
  \u2193
@1 <log_data >summary   (LLM: summarise the incident in plain English)
  \u2193
cpp <summary           (count words & sentences, print report)
```
Code
```cpp >log_data
#include <iostream>
int main() {
   std::cout <<
      "2026-03-11 02:14:33 ERROR  [auth-service]   Connection timeout after 30s (attempt 3/3)\n"
      "2026-03-11 02:14:35 ERROR  [db-pool]       Max connections reached (limit=100)\n"
      "2026-03-11 02:14:36 WARN   [api-gateway]    Retrying upstream /users endpoint\n"
```

```
    "2026-03-11 02:14:40 ERROR  [auth-service]   JWT validation failed: token expired\n"
    "2026-03-11 02:14:41 FATAL  [scheduler]      Worker thread panic: segmentation fault\n"
    "2026-03-11 02:14:42 INFO   [watchdog]       Restarting auth-service (pid 4872)\n";
  return 0;
}
```

<!--@1 <log_data >summary

You are a Site Reliability Engineer on-call.

The input contains a server log snippet.

Write a concise incident summary (2\u20133 sentences) suitable for a post-mortem report.

Cover: what failed, the cascade effect, and the automatic recovery action taken.

-->

```cpp <summary
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_summary"));
  std::string text((std::istreambuf_iterator<char>(f)),
                   std::istreambuf_iterator<char>());
  // count words
  std::istringstream ss(text);
  std::string tok;
  int words = 0;
  while (ss >> tok) ++words;
```

```
    // count sentences (. ! ?)
    int sentences = 0;
    for (char c : text) if (c == '.' || c == '!' || c == '?') ++sentences;
    std::cout << "=== LLM Incident Summary ===\n" << text << "\n"
              << "--- Meta: " << words << " words, ~"
              << sentences << " sentence(s) ---\n";
    return 0;
}
```

Key Patterns

- The LLM receives the **entire file content** of `log_data` as part of the prompt.
- C++ reads the LLM response file via `std::getenv("MSH_VAR_summary")` \u2014 same mechanism as any other variable.
- `std::istreambuf_iterator` is the idiomatic way to slurp a whole file into a `std::string` in C++.

API Error: messages.0: user messages must have non-empty content

2026-03-11 02:14:33 ERROR [auth-service] Connection timeout after 30s (attempt 3/3)

2026-03-11 02:14:35 ERROR [db-pool] Max connections reached (limit=100)

2026-03-11 02:14:36 WARN [api-gateway] Retrying upstream /users endpoint

2026-03-11 02:14:40 ERROR [auth-service] JWT validation failed: token expired

2026-03-11 02:14:41 FATAL [scheduler] Worker thread panic: segmentation fault

2026-03-11 02:14:42 INFO [watchdog] Restarting auth-service (pid 4872)

Incident Summary:

The auth-service experienced connection timeouts and JWT validation failures starting at 02:14:33, which cascaded to database connection pool exhaustion (100/100 connections) and API gateway retry loops. The situation escalated when the scheduler service suffered a fatal segmentation fault at 02:14:41, prompting the watchdog service to automatically restart the auth-service at 02:14:42. This incident demonstrates a classic failure cascade

where authentication issues led to resource exhaustion across multiple system components before triggering automated recovery mechanisms.

/home/igor > === LLM Incident Summary ===

Incident Summary:

The auth-service experienced connection timeouts and JWT validation failures starting at 02:14:33, which cascaded to database connection pool exhaustion (100/100 connections) and API gateway retry loops. The situation escalated when the scheduler service suffered a fatal segmentation fault at 02:14:41, prompting the watchdog service to automatically restart the auth-service at 02:14:42. This incident demonstrates a classic failure cascade where authentication issues led to resource exhaustion across multiple system components before triggering automated recovery mechanisms.

--- Meta: 76 words, ~3 sentence(s) ---

/home/igor > Sent to mshell (4189 bytes)

Received from GUI editor:

---

mshell C++ Edition

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

Concept: A single variable is produced once and independently consumed by multiple C++ blocks and an LLM \u2014 each performing a different analysis without modifying the shared source.

Use case: Multi-perspective analysis of a text corpus \u2014 one C++ block counts character-level stats, another computes word frequency, a third finds the longest word, and an LLM judges the writing style, all from the same input string.

Key concept: Multiple blocks can reference the same `<varname`. mshell never modifies a variable file on read \u2014 all consumers see the original, unchanged content.

Flow Diagram

```

cpp >text                    (produce the source text)
```

cpp <text cpp <text cpp <text @1 <text

(char stats)(word freq) (longest) (style judge)

```

## Code

```cpp >text
#include <iostream>
int main() {
  std::cout <<
    "The greatest glory in living lies not in never falling, "
    "but in rising every time we fall. "
    "The way to get started is to quit talking and begin doing. "
    "Life is what happens when you are busy making other plans."
    << std::endl;
  return 0;
}
```

```cpp <text
// Consumer 1: character-level statistics
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_text"));
  std::string text((std::istreambuf_iterator<char>(f)),
                   std::istreambuf_iterator<char>());
  int letters = 0, digits = 0, spaces = 0, punct = 0;
```
```

```
    for (char c : text) {
        if (std::isalpha(c))  ++letters;
        else if (std::isdigit(c)) ++digits;
        else if (std::isspace(c)) ++spaces;
        else if (std::ispunct(c)) ++punct;
    }
    std::cout << "[C++ Char Stats] letters=" << letters
              << " spaces=" << spaces
              << " punct=" << punct
              << " total=" << text.size() << "\n";
    return 0;
}
```

```cpp <text
// Consumer 2: top-3 most frequent words
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>
#include <cctype>
int main() {
    std::ifstream f(std::getenv("MSH_VAR_text"));
    std::string text((std::istreambuf_iterator<char>(f)),
                     std::istreambuf_iterator<char>());
```

```
    // normalise: strip punctuation, lowercase
    std::string clean;
    for (char c : text) clean += std::isalpha(c) ? std::tolower(c) : ' ';
    std::istringstream ss(clean);
    std::map<std::string, int> freq;
    std::string w;
    while (ss >> w) ++freq[w];
    std::vector<std::pair<int,std::string>> ranked;
    for (auto& p : freq) ranked.push_back({p.second, p.first});
    std::sort(ranked.rbegin(), ranked.rend());
    std::cout << "[C++ Word Freq] top words: ";
    for (int i = 0; i < std::min(3,(int)ranked.size()); ++i)
        std::cout << ranked[i].second << "(" << ranked[i].first << ") ";
    std::cout << "\n";
    return 0;
}
```

```cpp <text
// Consumer 3: longest word
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <cctype>
int main() {
    std::ifstream f(std::getenv("MSH_VAR_text"));
    std::string text((std::istreambuf_iterator<char>(f)),
```

```
                std::istreambuf_iterator<char>());
    std::string clean;
    for (char c : text) clean += std::isalpha(c) ? std::tolower(c) : ' ';
    std::istringstream ss(clean);
    std::string w, longest;
    while (ss >> w) if (w.size() > longest.size()) longest = w;
    std::cout << "[C++ Longest] \"" << longest
              << "\" (" << longest.size() << " chars)\n";
    return 0;
}
```

<!--@1 <text

The input is a passage of text.

In one sentence, describe its overall writing style (tone, vocabulary level, sentence length).

-->

Key Patterns

- Four independent blocks all read `MSH_VAR_text` \u2014 none of them writes back to it.

- Execution is sequential in document order, but the source file is never mutated.

- Each C++ binary is compiled independently; they share no state beyond the variable file.

The greatest glory in living lies not in never falling, but in rising every time we fall. The way to get started is to quit talking and begin doing. Life is what happens when you are busy making other plans.

[C++ Char Stats] letters=164 spaces=40 punct=4 total=208

[C++ Word Freq] top words: in(3) to(2) the(2)

[C++ Longest] "greatest" (8 chars)

The passage exhibits an inspirational, motivational tone using accessible vocabulary and concise, declarative sentences that deliver philosophical life advice in a direct, aphoristic style.

/home/igor > Sent to mshell (2674 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 4 LLM Code Generation Execute via Variable

Concept: An LLM generates executable C++ source code from a natural-language task description. A C++ "launcher" block reads the generated file path from `MSH_VAR_code`, compiles it with `g++`, and runs the resulting binary \u2014 all without any manual copy-paste.

Use case: On-the-fly algorithm generation \u2014 ask the model to implement a sorting algorithm, instantly compile and benchmark it.

Key concept: `MSH_VAR_code` is a **file path**. Compile with `cp MSH_VAR_code /tmp/gen.cpp && g++ -O2 -o /tmp/gen_bin /tmp/gen.cpp && /tmp/gen_bin` to turn LLM output into a running program.

Flow Diagram

```
cpp >task            (describe the coding task)
  \u2193
@1 <task >code       (LLM: write C++ source, no fences)
  \u2193
cpp <code            (compile with g++, run, print output)
```

Code

```cpp >task
#include <iostream>
int main() {
```

```
    std::cout <<
        "Write a C++ program that generates the first 15 Fibonacci numbers, "
        "prints them one per line, and then prints their sum. "
        "Use only the standard library. No classes needed."
        << std::endl;
    return 0;
}
```

<!--@1 <task >code

The input describes a C++ programming task.

Return ONLY valid C++ source code \u2014 no markdown fences, no explanation, no comments outside the code.

The code must compile cleanly with: g++ -std=c++17 -O2

-->

```cpp <code
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
int main() {
    const char* code_path = std::getenv("MSH_VAR_code");
    // copy to stable .cpp path \u2014 g++ needs a proper extension to recognise C++ source
    std::string cp_cmd = std::string("cp -- \"") + code_path + "\" /tmp/msh_p4_gen.cpp";
    std::system(cp_cmd.c_str());
    std::cout << "=== Generated C++ source ===\n";
    std::ifstream src("/tmp/msh_p4_gen.cpp");
    std::string line;
```

```
    while (std::getline(src, line)) std::cout << line << "\n";
    src.close();
    std::cout << "\n=== Compiling & Running ===\n";
    int ret = std::system(
        "g++ -std=c++17 -O2 -o /tmp/msh_p4_bin /tmp/msh_p4_gen.cpp 2>&1");
    if (ret != 0) {
        std::cerr << "Compilation failed\n";
        return 1;
    }
    std::system("/tmp/msh_p4_bin");
    return 0;
}
```

Key Patterns

- `MSH_VAR_code` holds the **path** to a temp file containing the LLM's text output.
- `std::system()` lets C++ drive the shell \u2014 compile then execute in one block.
- Always add `2>&1` to the compile command so errors are visible in the pipeline log.
- Use `-std=c++17 -O2` as the default compile flags; adjust per task.

Write a C++ program that generates the first 15 Fibonacci numbers, prints them one per line, and then prints their sum. Use only the standard library. No classes needed.

```
#include <iostream>
int main() {
    int a = 0, b = 1;
    int sum = a + b;
    std::cout << a << std::endl;
    std::cout << b << std::endl;
    for (int i = 2; i < 15; i++) {
```

```
        int next = a + b;
        std::cout << next << std::endl;
        sum += next;
        a = b;
        b = next;
    }
    std::cout << sum << std::endl;
    return 0;
}
/home/igor > === Generated C++ source ===
#include <iostream>
int main() {
    int a = 0, b = 1;
    int sum = a + b;
    std::cout << a << std::endl;
    std::cout << b << std::endl;
    for (int i = 2; i < 15; i++) {
        int next = a + b;
        std::cout << next << std::endl;
        sum += next;
        a = b;
        b = next;
    }
    std::cout << sum << std::endl;
    return 0;
}
=== Compiling & Running ===
```

0

1

1

2

3

5

8

13

21

34

55

89

144

233

377

986

/home/igor > Sent to mshell (3957 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 5 Two-LLM Review Chain

Concept: Model @1 generates an initial C++ solution. Model @2 reviews it for correctness, safety, and efficiency. Model @1 then receives both the original code and the review and produces an improved version. A final C++ block compiles and runs the result.

Use case: Automated C++ code quality improvement \u2014 generate a memory-safe implementation, have a second model critique it (looking for undefined behaviour, missing

bounds checks, etc.), then refine and execute.

Key concept: The refinement LLM call passes both `<code` and `<review` as inputs. When multiple `<invar` are present mshell labels each section with `[varname]:` in the prompt automatically.

Flow Diagram

```
cpp >task
@1 <task >code        (generate initial C++ implementation)
cpp <code             (print generated code)
@2 <code >review      (critique: correctness, UB, efficiency \u2014 2 sentences)
cpp <review           (print review)
@1 <code <review >improved   (refine based on review)
cpp <improved         (compile with g++, execute, print result)
```

Code

```cpp >task
#include <iostream>
int main() {
  std::cout <<
    "Write a C++ function bool is_palindrome(const std::string& s) "
    "that returns true if s reads the same forwards and backwards, "
    "ignoring case and non-alphabetic characters. "
    "Include a main() that tests it on at least 4 examples."
    << std::endl;
  return 0;
```

}

```

<!--@1 <task >code

The input describes a C++ programming task.

Return ONLY valid C++ source code \u2014 no markdown fences, no explanation.

Must compile with: g++ -std=c++17 -O2

-->

```cpp <code

#include <iostream>

#include <fstream>

#include <string>

int main() {

  std::ifstream f(std::getenv("MSH_VAR_code"));

  std::string line;

  std::cout << "=== Model @1 Generated ===\n";

  while (std::getline(f, line)) std::cout << line << "\n";

  return 0;

}

```

<!--@2 <code >review

You are a senior C++ engineer doing a code review.

The input is C++ source code.

In exactly 2 sentences: identify any bugs, undefined behaviour, missing edge cases,

or style issues, and suggest the most important fix.

-->

```cpp <review

#include <iostream>

```
#include <fstream>
#include <string>
int main() {
   std::ifstream f(std::getenv("MSH_VAR_review"));
   std::string line;
   std::cout << "=== Model @2 Review ===\n";
   while (std::getline(f, line)) std::cout << line << "\n";
   return 0;
}
```

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

The first input is C++ source code.

The second input is a code review.

Apply the review feedback and return ONLY the improved C++ source \u2014 no fences, no explanation.

Must compile with: g++ -std=c++17 -O2

-->

```cpp <improved
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
int main() {
   const char* path = std::getenv("MSH_VAR_improved");
   std::cout << "=== Final Improved Code ===\n";
   std::ifstream src(path);
   std::string line;
```

```
    while (std::getline(src, line)) std::cout << line << "\n";
    src.close();
    std::cout << "\n=== Compiling & Executing ===\n";
    // copy to stable .cpp path before compiling
    std::string cp_cmd = std::string("cp -- \"") + path + "\" /tmp/msh_p5_gen.cpp";
    std::system(cp_cmd.c_str());
    if (std::system("g++ -std=c++17 -O2 -o /tmp/msh_p5_bin /tmp/msh_p5_gen.cpp 2>&1")
!= 0) {
        std::cerr << "Compile failed\n"; return 1;
    }
    std::system("/tmp/msh_p5_bin");
    return 0;
}
```

Key Patterns

- **Role separation:** @1 = generator/fixer, @2 = critic. Neither model sees the other's system instructions.
- Multiple `<invar` on an LLM directive injects each variable with a `[varname]:` label, letting the model distinguish the original code from the review.
- The final C++ block uses `std::system()` to drive compilation \u2014 same pattern as Pattern 4.

Write a C++ function bool is_palindrome(const std::string& s) that returns true if s reads the same forwards and backwards, ignoring case and non-alphabetic characters. Include a main() that tests it on at least 4 examples.

```
#include <iostream>
#include <string>
#include <cctype>
bool is_palindrome(const std::string& s) {
```

```
    int left = 0;
    int right = s.length() - 1;
    while (left < right) {
        while (left < right && !std::isalpha(s[left])) {
            left++;
        }
        while (left < right && !std::isalpha(s[right])) {
            right--;
        }
        if (std::tolower(s[left]) != std::tolower(s[right])) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
int main() {
    std::cout << is_palindrome("A man, a plan, a canal: Panama") << std::endl;
    std::cout << is_palindrome("race a car") << std::endl;
    std::cout << is_palindrome("Madam") << std::endl;
    std::cout << is_palindrome("No 'x' in Nixon") << std::endl;
    std::cout << is_palindrome("Was it a car or a cat I saw?") << std::endl;
    return 0;
}
=== Model @1 Generated ===
#include <iostream>
```

```
#include <string>
#include <cctype>
bool is_palindrome(const std::string& s) {
    int left = 0;
    int right = s.length() - 1;
    while (left < right) {
        while (left < right && !std::isalpha(s[left])) {
            left++;
        }
        while (left < right && !std::isalpha(s[right])) {
            right--;
        }
        if (std::tolower(s[left]) != std::tolower(s[right])) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
int main() {
    std::cout << is_palindrome("A man, a plan, a canal: Panama") << std::endl;
    std::cout << is_palindrome("race a car") << std::endl;
    std::cout << is_palindrome("Madam") << std::endl;
    std::cout << is_palindrome("No 'x' in Nixon") << std::endl;
    std::cout << is_palindrome("Was it a car or a cat I saw?") << std::endl;
    return 0;
```

}

`std::isalpha` and `std::tolower` have undefined behavior when passed a `char` that is negative (other than EOF), so you should cast to `unsigned char` (e.g., `std::isalpha(static_cast<unsigned char>(s[left]))`). Also, consider using `std::size_t` for indices to avoid signed/unsigned mismatch warnings, but the most important fix is correcting the calls to the cctype functions.

=== Model @2 Review ===

`std::isalpha` and `std::tolower` have undefined behavior when passed a `char` that is negative (other than EOF), so you should cast to `unsigned char` (e.g., `std::isalpha(static_cast<unsigned char>(s[left]))`). Also, consider using `std::size_t` for indices to avoid signed/unsigned mismatch warnings, but the most important fix is correcting the calls to the cctype functions.

```
#include <iostream>

#include <string>

#include <cctype>

bool is_palindrome(const std::string& s) {

   std::size_t left = 0;

   std::size_t right = s.length() - 1;

   while (left < right) {

      while (left < right && !std::isalpha(static_cast<unsigned char>(s[left]))) {

         left++;

      }

      while (left < right && !std::isalpha(static_cast<unsigned char>(s[right]))) {

         right--;

      }

      if (std::tolower(static_cast<unsigned char>(s[left])) !=
std::tolower(static_cast<unsigned char>(s[right]))) {

         return false;

      }

      left++;

      right--;
```

```
    }
    return true;
}
int main() {
    std::cout << is_palindrome("A man, a plan, a canal: Panama") << std::endl;
    std::cout << is_palindrome("race a car") << std::endl;
    std::cout << is_palindrome("Madam") << std::endl;
    std::cout << is_palindrome("No 'x' in Nixon") << std::endl;
    std::cout << is_palindrome("Was it a car or a cat I saw?") << std::endl;
    return 0;
}
/home/igor > === Final Improved Code ===
#include <iostream>
#include <string>
#include <cctype>
bool is_palindrome(const std::string& s) {
    std::size_t left = 0;
    std::size_t right = s.length() - 1;
    while (left < right) {
        while (left < right && !std::isalpha(static_cast<unsigned char>(s[left]))) {
            left++;
        }
        while (left < right && !std::isalpha(static_cast<unsigned char>(s[right]))) {
            right--;
        }
        if (std::tolower(static_cast<unsigned char>(s[left])) !=
std::tolower(static_cast<unsigned char>(s[right]))) {
```

```
            return false;
        }
        left++;
        right--;
    }
    return true;
}
int main() {
    std::cout << is_palindrome("A man, a plan, a canal: Panama") << std::endl;
    std::cout << is_palindrome("race a car") << std::endl;
    std::cout << is_palindrome("Madam") << std::endl;
    std::cout << is_palindrome("No 'x' in Nixon") << std::endl;
    std::cout << is_palindrome("Was it a car or a cat I saw?") << std::endl;
    return 0;
}
=== Compiling & Executing ===
1
0
1
1
1
----------------------------------------------------------------------
/home/igor > Sent to mshell (2693 bytes)
Received from GUI editor:
---------------------------------------
# mshell C++ Edition
# Pattern 6  Parallel 3-Model Query
```

Concept: The same C++-generated question is sent to all three LLM models independently. Each model stores its answer in a separate variable. A final C++ block collects and formats all three responses side-by-side.

Use case: Ensemble design advice \u2014 ask three models to recommend the best C++ data structure for a given problem, then compare their reasoning.

Key concept: Three separate LLM directive blocks with `@1`, `@2`, `@3` all read the same `<question` and write to distinct `>ans1`, `>ans2`, `>ans3` output variables. Execution is sequential in document order but each model is independent.

Flow Diagram

```
cpp >question
 \u2193       \u2193        \u2193
@1 <question  @2 <question  @3 <question
>ans1       >ans2       >ans3
 \u2193       \u2193        \u2193
cpp <ans1 <ans2 <ans3   (print all three side-by-side)
```

Code

```cpp >question
#include <iostream>
int main() {
   std::cout <<
      "In modern C++17, what is the single most important container or "
      "data structure a developer should deeply understand, and why? "
      "Answer in exactly two sentences."
```

```
      << std::endl;
   return 0;
}
```

<!--@1 <question >ans1

Answer the question in exactly two sentences. Focus on performance characteristics.

-->

<!--@2 <question >ans2

Answer the question in exactly two sentences. Focus on practical everyday usage.

-->

<!--@3 <question >ans3

Answer the question in exactly two sentences. Focus on memory safety and modern best practices.

-->

```cpp <ans1 <ans2 <ans3
#include <iostream>
#include <fstream>
#include <string>
// helper: read entire file to string
std::string slurp(const char* env_key) {
   std::ifstream f(std::getenv(env_key));
   return std::string((std::istreambuf_iterator<char>(f)),
                      std::istreambuf_iterator<char>());
}
int main() {
   auto a1 = slurp("MSH_VAR_ans1");
   auto a2 = slurp("MSH_VAR_ans2");
```

```
    auto a3 = slurp("MSH_VAR_ans3");
    std::string sep(60, '-');
    std::cout << "=== Model @1 (Performance) ===\n" << a1 << "\n"
              << sep << "\n"
              << "=== Model @2 (Practical) ===\n"   << a2 << "\n"
              << sep << "\n"
              << "=== Model @3 (Safety / Modern) ===\n" << a3 << "\n";
    return 0;
}
```

Key Patterns

- All three LLM blocks are independent \u2014 each reads the original `question` file, writes
  its own output variable.
- The collecting C++ block uses a `slurp()` helper to read multiple variable files cleanly.
- To run them truly in parallel, add `async` to each directive and add an `await=` barrier
  (see Pattern 12).

In modern C++17, what is the single most important container or data structure a developer should deeply understand, and why? Answer in exactly two sentences.

The `std::vector` is the single most important container in modern C++17 because it provides cache-friendly contiguous memory layout that maximizes CPU performance, while offering amortized O(1) insertions and O(1) random access that outperforms most alternatives in real-world scenarios. Its combination of memory efficiency, iterator stability during most operations, and compatibility with algorithms makes it the default choice that should only be replaced when specific requirements (like frequent front insertions) explicitly demand different performance characteristics.

`std::vector` is the single most important container to deeply understand because it is the default, go-to choice for storing sequences of elements, with contiguous memory, excellent cache locality, and amortized constant-time `push_back` and random access. Mastering `std::vector`\u2014including how it grows, how capacity vs. size works, and how it interacts with iterators, algorithms, and move semantics\u2014directly improves performance, safety, and clarity in most everyday C++17 code.

The `std::vector` is the single most important container because it enforces memory safety through RAII, automatically managing dynamic memory to prevent leaks, dangling pointers, and buffer overflows. Modern C++17 practices like `emplace_back` and structured bindings maximize its efficiency while ensuring exception safety and adherence to value semantics.

/home/igor > === Model @1 (Performance) ===

The `std::vector` is the single most important container in modern C++17 because it provides cache-friendly contiguous memory layout that maximizes CPU performance, while offering amortized O(1) insertions and O(1) random access that outperforms most alternatives in real-world scenarios. Its combination of memory efficiency, iterator stability during most operations, and compatibility with algorithms makes it the default choice that should only be replaced when specific requirements (like frequent front insertions) explicitly demand different performance characteristics.

=== Model @2 (Practical) ===

`std::vector` is the single most important container to deeply understand because it is the default, go-to choice for storing sequences of elements, with contiguous memory, excellent cache locality, and amortized constant-time `push_back` and random access. Mastering `std::vector`\u2014including how it grows, how capacity vs. size works, and how it interacts with iterators, algorithms, and move semantics\u2014directly improves performance, safety, and clarity in most everyday C++17 code.

=== Model @3 (Safety / Modern) ===

The `std::vector` is the single most important container because it enforces memory safety through RAII, automatically managing dynamic memory to prevent leaks, dangling pointers, and buffer overflows. Modern C++17 practices like `emplace_back` and structured bindings maximize its efficiency while ensuring exception safety and adherence to value semantics.

/home/igor > Sent to mshell (4293 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 7 Evaluator-Optimizer Loop

Concept: A generator LLM produces a C++ solution. An evaluator LLM reviews it and

returns exactly `ACCEPTED` or `REJECTED: <reason>`. The loop repeats \u2014 passing both the

previous code and the verdict back to the generator \u2014 until accepted or the iteration cap

is reached. The final accepted code is compiled and executed by a C++ block.

Use case: Automated iterative refinement of a C++ algorithm until it passes a strict

quality gate (correctness, no raw pointers, must use RAII).

Key concept: `<!--@loop max=N until=verdict:ACCEPTED-->` wraps the

generator-evaluator pair. The loop exits early on the acceptance condition or after N

iterations as a safety cap.

Flow Diagram

```
cpp >task
 [LOOP max=3 until=verdict:ACCEPTED]
  @1 <task >code        (generate / refine C++ code)
  cpp <code             (print generated code)
  @2 <code >verdict     (evaluate: ACCEPTED or REJECTED: reason)
  cpp <verdict          (print verdict)
[END_LOOP]
cpp <code               (compile final code, run, print output)
```

Code

```cpp >task
#include <iostream>
int main() {
   std::cout <<
      "Write a C++ function std::vector<int> sieve(int n) that returns all "
      "prime numbers up to n using the Sieve of Eratosthenes. "
```

```
        "Requirements: use RAII (no raw new/delete), mark functions constexpr "
        "where possible, include a main() that prints primes up to 50."
        << std::endl;
    return 0;
}
```

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

<!--@1 <task >code

The input describes a C++ programming task with strict requirements.

Return ONLY valid C++ source \u2014 no fences, no explanation.

Must compile with: g++ -std=c++17 -O2

If a previous attempt was rejected, fix the issues mentioned.

-->

```cpp <code
#include <iostream>
#include <fstream>
#include <string>
int main() {
    std::cout << "=== Generated Code ===\n";
    std::ifstream f(std::getenv("MSH_VAR_code"));
    std::string line;
    while (std::getline(f, line)) std::cout << line << "\n";
    return 0;
}
```

<!--@2 <code >verdict

You are a strict C++ code reviewer.

The input is C++ source code.

Check: (1) correct algorithm, (2) RAII / no raw new/delete, (3) compiles with g++ -std=c++17 -O2.

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

If any check fails, reply with: REJECTED: <one sentence describing the first issue>

-->

```cpp <verdict
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_verdict"));
  std::string verdict((std::istreambuf_iterator<char>(f)),
                      std::istreambuf_iterator<char>());
  // extract first word (ACCEPTED or REJECTED)
  std::istringstream ss(verdict);
  std::string first_word;
  ss >> first_word;
  std::cout << "=== Verdict: " << first_word << " ===\n";
  // last stdout line must equal the verdict word for LOOP condition check
  std::cout << first_word << "\n";
  return 0;
}
```

<!--@end_loop-->

```cpp <code
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
int main() {
    const char* path = std::getenv("MSH_VAR_code");
    std::cout << "=== Final Accepted Code ===\n";
    std::ifstream src(path);
    std::string line;
    while (std::getline(src, line)) std::cout << line << "\n";
    src.close();
    std::cout << "\n=== Compiling & Running ===\n";
    // copy to stable .cpp path before compiling
    std::string cp_cmd = std::string("cp -- \"") + path + "\" /tmp/msh_p7_gen.cpp";
    std::system(cp_cmd.c_str());
    if (std::system("g++ -std=c++17 -O2 -o /tmp/msh_p7_bin /tmp/msh_p7_gen.cpp 2>&1")
!= 0) {
        std::cerr << "Compile failed\n"; return 1;
    }
    std::system("/tmp/msh_p7_bin");
    return 0;
}
```

Key Patterns

- The loop variable `verdict` must contain exactly `ACCEPTED` to trigger early exit \u2014 prompt the evaluator model very strictly.
- On each loop iteration the generator sees the same `<task` variable, so it can incorporate

`REJECTED:` feedback only if you also pass `<verdict` to it (add `<verdict` to the @1 directive for self-correction).

- `max=3` is a safety cap \u2014 always set it to prevent infinite loops.

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

Write a C++ function std::vector<int> sieve(int n) that returns all prime numbers up to n using the Sieve of Eratosthenes. Requirements: use RAII (no raw new/delete), mark functions constexpr where possible, include a main() that prints primes up to 50.

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

```
#include <iostream>
#include <vector>
std::vector<int> sieve(int n) {
  if (n < 2) return {};
  std::vector<bool> is_prime(n + 1, true);
  is_prime[0] = is_prime[1] = false;
  for (int i = 2; i * i <= n; ++i) {
    if (is_prime[i]) {
      for (int j = i * i; j <= n; j += i) {
        is_prime[j] = false;
      }
    }
  }
  std::vector<int> primes;
  for (int i = 2; i <= n; ++i) {
    if (is_prime[i]) {
      primes.push_back(i);
    }
  }
  return primes;
```

```
}
int main() {
    auto primes = sieve(50);
    for (int prime : primes) {
        std::cout << prime << " ";
    }
    std::cout << std::endl;
    return 0;
}
=== Generated Code ===
#include <iostream>
#include <vector>
std::vector<int> sieve(int n) {
    if (n < 2) return {};
    std::vector<bool> is_prime(n + 1, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i * i <= n; ++i) {
        if (is_prime[i]) {
            for (int j = i * i; j <= n; j += i) {
                is_prime[j] = false;
            }
        }
    }
    std::vector<int> primes;
    for (int i = 2; i <= n; ++i) {
        if (is_prime[i]) {
            primes.push_back(i);
```

```
        }
    }
    return primes;
}
int main() {
    auto primes = sieve(50);
    for (int prime : primes) {
        std::cout << prime << " ";
    }
    std::cout << std::endl;
    return 0;
}
ACCEPTED
[loop] Exiting loop after 1 iteration(s). reason: until condition met
=== Verdict: ACCEPTED ===
ACCEPTED
/home/igor > === Final Accepted Code ===
#include <iostream>
#include <vector>
std::vector<int> sieve(int n) {
    if (n < 2) return {};
    std::vector<bool> is_prime(n + 1, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i * i <= n; ++i) {
        if (is_prime[i]) {
            for (int j = i * i; j <= n; j += i) {
                is_prime[j] = false;
```

```
            }
        }
    }
    std::vector<int> primes;
    for (int i = 2; i <= n; ++i) {
        if (is_prime[i]) {
            primes.push_back(i);
        }
    }
    return primes;
}
int main() {
    auto primes = sieve(50);
    for (int prime : primes) {
        std::cout << prime << " ";
    }
    std::cout << std::endl;
    return 0;
}
=== Compiling & Running ===
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
---------------------------------------------------------------------
/home/igor > Sent to mshell (4189 bytes)
Received from GUI editor:
---------------------------------------
# mshell C++ Edition
# Pattern 8 Multi-Stage C++ + Multi-Model Pipeline
```

Concept: Replaces multi-language pipelines with multiple C++ stages, each acting as a specialist. C++ generates raw data, a second C++ block computes statistics, Model @1 provides a narrative analysis, Model @2 compresses the analysis to a headline, and a final C++ block collects everything and formats a structured report.

Use case: Automated benchmark reporting \u2014 C++ produces timing data, C++ derives statistics, two LLMs add interpretation and a tweet-sized summary, C++ formats the final report.

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

Flow Diagram

```
cpp >raw_data          (generate 12 random latency measurements in ms)
cpp <raw_data >stats    (compute mean, median, p95, min, max)
@1 <stats >analysis     (LLM: narrative interpretation)
@2 <analysis >headline  (LLM: compress to \u226410-word headline)
cpp <raw_data <stats <analysis <headline   (print full structured report)
```

Code

```cpp >raw_data
#include <iostream>
int main() {
  // deterministic pseudo-latencies (ms), comma-separated
  int latencies[] = {12, 45, 8, 102, 23, 67, 15, 88, 34, 201, 19, 56};
  for (int i = 0; i < 12; ++i) {
    std::cout << latencies[i];
    if (i < 11) std::cout << ",";
```

```
    }
    std::cout << "\n";
    return 0;
}
```

```cpp <raw_data >stats
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <algorithm>
#include <numeric>
int main() {
    std::ifstream f(std::getenv("MSH_VAR_raw_data"));
    std::string raw;
    std::getline(f, raw);
    std::vector<double> v;
    std::stringstream ss(raw);
    std::string tok;
    while (std::getline(ss, tok, ',')) v.push_back(std::stod(tok));
    std::sort(v.begin(), v.end());
    double sum   = std::accumulate(v.begin(), v.end(), 0.0);
    double mean  = sum / v.size();
    double med   = v.size() % 2 == 0
                 ? (v[v.size()/2-1] + v[v.size()/2]) / 2.0
                 : v[v.size()/2];
```

```
    double p95   = v[(int)(v.size() * 0.95)];
    std::cout << "count=" << v.size()
              << " min="  << v.front()
              << " max="  << v.back()
              << " mean=" << mean
              << " median=" << med
              << " p95="  << p95
              << "\n";
    return 0;
}
```

<!--@1 <stats >analysis

The input contains latency statistics for a server benchmark (values in milliseconds).

Write one paragraph (3 sentences) interpreting what these numbers mean for production

readiness \u2014 mention whether the tail latency is acceptable and what the spread suggests.

-->

<!--@2 <analysis >headline

The input is a paragraph of technical analysis.

Compress it into a single headline of at most 10 words suitable for a dashboard alert.

Reply with ONLY the headline, no punctuation at the end.

-->

```cpp <raw_data <stats <analysis <headline
#include <iostream>
#include <fstream>
#include <string>
std::string slurp(const char* key) {
```

```
    std::ifstream f(std::getenv(key));
    return {std::istreambuf_iterator<char>(f), {}};
}
int main() {
    auto raw      = slurp("MSH_VAR_raw_data");
    auto stats    = slurp("MSH_VAR_stats");
    auto analysis = slurp("MSH_VAR_analysis");
    auto headline = slurp("MSH_VAR_headline");
    std::string sep(60, '=');
    std::cout << sep << "\n"
              << "BENCHMARK REPORT\n"
              << sep << "\n"
              << "Raw data (ms) : " << raw
              << "Statistics    : " << stats
              << "\nAnalysis:\n"    << analysis
              << "\nHeadline: >>> " << headline << " <<<\n"
              << sep << "\n";
    return 0;
}
```

Key Patterns

- C++ replaces every other language: data generation, statistics, and report formatting are all separate compiled binaries communicating via mshell variables.
- `slurp()` helper avoids repetitive file-reading boilerplate when a block needs many input variables.
- LLM directives slot in between C++ stages just like any other block.

12,45,8,102,23,67,15,88,34,201,19,56

count=12 min=8 max=201 mean=55.8333 median=39.5 p95=201

The mean latency of 55.8ms with a median of 39.5ms indicates reasonable typical performance, but the significant difference between these values suggests a right-skewed distribution with some slower requests pulling the average upward. The p95 latency of 201ms represents concerning tail behavior, as it means 5% of users experience response times over 200ms, which could negatively impact user experience and may not meet typical SLA requirements for web applications. The wide spread from 8ms to 201ms (a 25x difference between min and max) indicates inconsistent performance that would benefit from investigation into potential bottlenecks, resource contention, or optimization opportunities before production deployment.

High p95 latency and inconsistent response times indicate performance issues

==========

BENCHMARK REPORT

==========

Raw data (ms) : 12,45,8,102,23,67,15,88,34,201,19,56

Statistics : count=12 min=8 max=201 mean=55.8333 median=39.5 p95=201

Analysis:

The mean latency of 55.8ms with a median of 39.5ms indicates reasonable typical performance, but the significant difference between these values suggests a right-skewed distribution with some slower requests pulling the average upward. The p95 latency of 201ms represents concerning tail behavior, as it means 5% of users experience response times over 200ms, which could negatively impact user experience and may not meet typical SLA requirements for web applications. The wide spread from 8ms to 201ms (a 25x difference between min and max) indicates inconsistent performance that would benefit from investigation into potential bottlenecks, resource contention, or optimization opportunities before production deployment.

Headline: >>> High p95 latency and inconsistent response times indicate performance issues <<<

==========

/home/igor > Sent to mshell (5536 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 9 Routing (LLM Classifies Conditional C++ Branch Executes)

Concept: An LLM classifies a C++-generated input into one of several categories. Each

downstream C++ block carries an `if=route:VALUE` condition \u2014 only the matching block

runs; all others are silently skipped.

Use case: Smart computation dispatcher \u2014 input can be a sorting request, a math

formula, or a text transformation; the LLM routes it to the correct C++ handler automatically.

Key concept: `if=varname:expected_value` on any block makes it conditional. The LLM

classifier must return a single, predictable word \u2014 prompt discipline is critical.

Flow Diagram

```

cpp >input

@1 <input >route       (classify: SORT / MATH / TEXT)

cpp <input if=route:SORT   (run sort algorithm, print sorted result)

cpp <input if=route:MATH   (evaluate the math expression, print result)

cpp <input if=route:TEXT   (reverse & uppercase the string, print)

cpp <route             (print which branch was taken)

```

Three separate test runs are shown (one per category).

Code Test 1: SORT

```cpp >input1

#include <iostream>

int main() {

   std::cout << "sort these integers: 9 3 7 1 5 8 2 6 4" << std::endl;

   return 0;

}

```

<!--@1 <input1 >route1

Classify the task described in the input into exactly one word: SORT, MATH, or TEXT.

Reply with ONLY that one word, nothing else.

-->

```cpp <input1 if=route1:SORT
// SORT branch: extract integers and sort them
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <algorithm>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_input1"));
  std::string line; std::getline(f, line);
  // extract digits from the sentence
  std::istringstream ss(line);
  std::string tok;
  std::vector<int> nums;
  while (ss >> tok) {
    try { nums.push_back(std::stoi(tok)); } catch(...) {}
  }
  std::sort(nums.begin(), nums.end());
  std::cout << "[SORT branch] sorted: ";
  for (int n : nums) std::cout << n << " ";
  std::cout << "\n";
  return 0;
```

}
```

```cpp <input1 if=route1:MATH
#include <iostream>
int main() { std::cout << "[MATH branch \u2014 unexpected for test 1]\n"; return 0; }
```

```cpp <input1 if=route1:TEXT
#include <iostream>
int main() { std::cout << "[TEXT branch \u2014 unexpected for test 1]\n"; return 0; }
```

```cpp <route1
#include <iostream>
#include <fstream>
#include <string>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_route1"));
  std::string r; std::getline(f, r);
  std::cout << "Test1 routed to: " << r << "\n\n";
  return 0;
}
```

Code Test 2: MATH

```cpp >input2
#include <iostream>
int main() {
  std::cout << "compute the value of: 2 to the power of 10" << std::endl;
  return 0;

}
```
<!--@1 <input2 >route2
Classify the task described in the input into exactly one word: SORT, MATH, or TEXT.
Reply with ONLY that one word, nothing else.
-->
```cpp <input2 if=route2:SORT
#include <iostream>
int main() { std::cout << "[SORT branch \u2014 unexpected for test 2]\n"; return 0; }
```
```cpp <input2 if=route2:MATH
// MATH branch: compute 2^10
#include <iostream>
#include <cmath>
int main() {
    long long result = (long long)std::pow(2, 10);
    std::cout << "[MATH branch] 2^10 = " << result << "\n";
    return 0;
}
```
```cpp <input2 if=route2:TEXT
#include <iostream>
int main() { std::cout << "[TEXT branch \u2014 unexpected for test 2]\n"; return 0; }
```
```cpp <route2
#include <iostream>
#include <fstream>

```
#include <string>
int main() {
   std::ifstream f(std::getenv("MSH_VAR_route2"));
   std::string r; std::getline(f, r);
   std::cout << "Test2 routed to: " << r << "\n\n";
   return 0;
}
```

Code Test 3: TEXT

```cpp >input3
#include <iostream>
int main() {
   std::cout << "transform this string: hello mshell world" << std::endl;
   return 0;
}
```

<!--@1 <input3 >route3
Classify the task described in the input into exactly one word: SORT, MATH, or TEXT.
Reply with ONLY that one word, nothing else.
-->

```cpp <input3 if=route3:SORT
#include <iostream>
int main() { std::cout << "[SORT branch \u2014 unexpected for test 3]\n"; return 0; }
```

```cpp <input3 if=route3:MATH
#include <iostream>
int main() { std::cout << "[MATH branch \u2014 unexpected for test 3]\n"; return 0; }
```

```

```cpp <input3 if=route3:TEXT
// TEXT branch: reverse & uppercase the input string
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <cctype>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_input3"));
  std::string line; std::getline(f, line);
  // extract the substring after ": "
  auto pos = line.find(": ");
  std::string s = (pos != std::string::npos) ? line.substr(pos + 2) : line;
  std::reverse(s.begin(), s.end());
  for (char& c : s) c = std::toupper(c);
  std::cout << "[TEXT branch] reversed+upper: " << s << "\n";
  return 0;
}
```

```cpp <route3
#include <iostream>
#include <fstream>
#include <string>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_route3"));
  std::string r; std::getline(f, r);

```
    std::cout << "Test3 routed to: " << r << "\n";
    return 0;
}
```

Key Patterns

- `if=route:SORT` \u2014 the block runs only when the variable `route` contains exactly `SORT`.
- Skipped blocks produce no output and no variable writes \u2014 as if they don't exist.
- Always include "else" stubs (with unexpected-branch messages) during development to catch mis-classifications.

---

sort these integers: 9 3 7 1 5 8 2 6 4

SORT

[SORT branch] sorted: 1 2 3 4 5 6 7 8 9

cc1plus: fatal error: /tmp/mshell_code_316797.cpp: No such file or directory

compilation terminated.

compute the value of: 2 to the power of 10

MATH

[MATH branch] 2^10 = 1024

Test2 routed to: MATH

transform this string: hello mshell world

TEXT

[TEXT branch] reversed+upper: DLROW LLEHSM OLLEH

/home/igor > Test3 routed to: TEXT

/home/igor > Sent to mshell (4440 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 10 Full C++ Pipeline (All Patterns Combined)

Concept: A complete end-to-end showcase: C++ generates prime numbers, a second C++

block derives statistics, two LLMs analyse and poeticise in parallel (async), a C++

`await=` barrier synchronises them, a third LLM synthesises the results, and a final C++

block renders the output in a decorative frame.

Use case: Reference template demonstrates that all building blocks (sequential

stages, fan-out, async LLM, await barrier, synthesis) compose naturally in a single mshell

document using only C++.

Key concept: There is no special "combine patterns" syntax. Every block simply reads

named variables the document structure IS the execution graph.

Flow Diagram

```
cpp >raw_data          (Sieve: first 10 primes)
cpp <raw_data >stats    (min, max, mean, sum)
@1 <stats >analysis async    @2 <raw_data >poem async
cpp await=analysis,poem
cpp <analysis <poem     (print both)
@1 <analysis <poem >combined   (synthesise into one elegant sentence)
cpp <combined          (render in decorative C++ frame)
```

Code

```cpp >raw_data
#include <iostream>
```

```
#include <vector>
int main() {
   std::vector<int> primes;
   for (int n = 2; (int)primes.size() < 10; ++n) {
      bool ok = true;
      for (int i = 2; i * i <= n; ++i) if (n % i == 0) { ok = false; break; }
      if (ok) primes.push_back(n);
   }
   for (int i = 0; i < (int)primes.size(); ++i) {
      std::cout << primes[i];
      if (i + 1 < (int)primes.size()) std::cout << " ";
   }
   std::cout << "\n";
   return 0;
}
```

```cpp <raw_data >stats
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <numeric>
#include <algorithm>
int main() {
   std::ifstream f(std::getenv("MSH_VAR_raw_data"));
   std::string raw; std::getline(f, raw);
```

```
    std::istringstream ss(raw);
    std::vector<int> v;
    int x; while (ss >> x) v.push_back(x);
    double mean = std::accumulate(v.begin(), v.end(), 0.0) / v.size();
    std::cout << "count=" << v.size()
              << " min="  << *std::min_element(v.begin(), v.end())
              << " max="  << *std::max_element(v.begin(), v.end())
              << " sum="  << std::accumulate(v.begin(), v.end(), 0)
              << " mean=" << mean
              << "\n";
    return 0;
}
```

<!--@1 <stats >analysis async

The input contains statistics about the first 10 prime numbers.

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

-->

<!--@2 <raw_data >poem async

The input is a list of prime numbers.

Write a 2-line poem (rhyming couplet) celebrating the beauty of primes.

-->

```cpp await=analysis,poem
// synchronisation barrier \u2014 no code needed, just the await= attribute
#include <iostream>
int main() {
    std::cout << "[await] both LLM jobs complete\n";
    return 0;
```

}

```

```cpp <analysis <poem
#include <iostream>
#include <fstream>
#include <string>
std::string slurp(const char* key) {
  std::ifstream f(std::getenv(key));
  return {std::istreambuf_iterator<char>(f), {}};
}
int main() {
  std::cout << "=== Analysis ===\n" << slurp("MSH_VAR_analysis") << "\n"
        << "=== Poem ===\n"    << slurp("MSH_VAR_poem")    << "\n";
  return 0;
}
```

<!--@1 <analysis <poem >combined
The first input is a mathematical analysis. The second is a poem.
Combine both perspectives into one elegant sentence that is both precise and beautiful.
-->
```cpp <combined
#include <iostream>
#include <fstream>
#include <string>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_combined"));
  std::string text((std::istreambuf_iterator<char>(f)), {});

```
    // trim trailing newline
    while (!text.empty() && text.back() == '\n') text.pop_back();
    std::string border(text.size() + 4, '*');
    std::cout << border << "\n"
              << "* " << text << " *\n"
              << border << "\n";
    return 0;
}
```

Key Patterns

- `async` on an LLM directive launches a background job; `await=analysis,poem` blocks until both output variables are written.
- Total time = max(time(@1), time(@2)), not the sum.
- The synthesising LLM (`@1 <analysis <poem`) receives both variables with `[varname]:` labels injected automatically.
- No special "combine" syntax \u2014 variable names are the only wiring mechanism.

2 3 5 7 11 13 17 19 23 29

count=10 min=2 max=29 sum=129 mean=12.9

[async llm] Launched PID 320368 \u2192 var=analysis (model @1)

[async llm] Launched PID 320369 \u2192 var=poem (model @2)

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

[async] Waiting for PID 320368 (var=analysis)...

[async] PID 320368 done (var=analysis)

The data reveals that among the first 10 primes, the distribution shows moderate clustering in the lower range (minimum of 2) but exhibits significant spread as primes become more sparse at higher values (maximum of 29), with the mean of 12.9 demonstrating how the increasing gaps between consecutive primes affect the overall average.

=== Analysis ===

The data reveals that among the first 10 primes, the distribution shows moderate clustering in the lower range (minimum of 2) but exhibits significant spread as primes become more sparse at higher values (maximum of 29), with the mean of 12.9 demonstrating how the increasing gaps between consecutive primes affect the overall average.

=== Poem ===

In quiet rows they stand, primes woven through our time,

Indivisible whispers of order and mystery sublime

Like scattered stars across the mathematical sky, the first ten primes whisper their ancient secrets through widening gaps\u2014from 2's gentle beginning to 29's distant light\u2014their mean of 12.9 a tender measure of how beauty grows more precious as it becomes more rare.

/home/igor > **********

* Like scattered stars across the mathematical sky, the first ten primes whisper their ancient secrets through widening gaps\u2014from 2's gentle beginning to 29's distant light\u2014their mean of 12.9 a tender measure of how beauty grows more precious as it becomes more rare. *

****

/home/igor > Sent to mshell (2590 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 11 MShell Node with Multiple Models (C++ Data Sources)

Concept: Native `mshell` code blocks call LLM models using `ollama1`/`ollama2`

commands, with C++ blocks producing and consuming the surrounding variables.

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

directives \u2014 for example, when you need to build the prompt dynamically inside the mshell

block itself, or combine mshell commands (`print`, `eval`) with model calls.

Key concept: `mshell` blocks support the same `<in_vars` and `>out_var` variable

system as C++ blocks. Input variables are loaded into the mshell variable table before

execution, and stdout is captured into the output variable.

Flow Diagram

```
cpp >topic            (produce topic string)
cpp >constraints      (produce constraint string)
mshell <topic <constraints >explanation   (ollama1: explain topic)
mshell <explanation >keywords           (ollama2: extract keywords)
cpp <topic <explanation <keywords       (format final report)
```

Code

```cpp >topic
#include <iostream>
int main() {
    std::cout << "move semantics and rvalue references in C++11" << std::endl;
    return 0;
}
```

```cpp >constraints
#include <iostream>
int main() {
    std::cout << "beginner-friendly, max 2 sentences, include a one-line analogy" << std::endl;
    return 0;
}
```

```mshell <topic <constraints >explanation
ollama1 "Explain $topic. Style: $constraints"
```

```

```mshell <explanation >keywords

ollama2 "Extract exactly 4 technical keywords from this text: $explanation. Reply with only the 4 words, comma-separated."

```

```cpp <topic <explanation <keywords

#include <iostream>

#include <fstream>

#include <string>

std::string slurp(const char* key) {

  std::ifstream f(std::getenv(key));

  return {std::istreambuf_iterator<char>(f), {}};

}

int main() {

  std::cout << "=== Topic ===\n"      << slurp("MSH_VAR_topic")

        << "\n=== Explanation ===\n" << slurp("MSH_VAR_explanation")

        << "\n=== Keywords ===\n"    << slurp("MSH_VAR_keywords") << "\n";

  return 0;

}

```

Key Patterns

- `mshell` blocks interpolate variable contents with `$varname` syntax directly in the `ollama1`/`ollama2` command string.
- C++ blocks handle structured data production and final formatting; mshell blocks handle the AI calls.
- `ollama1exec`, `ollama2exec`, `ollama3exec` are internal implementations for running LLM-generated code directly use them when the model output is itself executable.
```

move semantics and rvalue references in C++11

beginner-friendly, max 2 sentences, include a one-line analogy

Move semantics in C++11 allows you to efficiently "steal" resources from temporary objects (rvalue references, marked with &&) instead of making expensive copies, using std::move() to transfer ownership of data like moving contents between containers. Think of it like moving houses: instead of laboriously copying every item to your new home (copy semantics), you just pick up your furniture and relocate it (move semantics) - much faster and less wasteful.Move semantics, rvalue references, std::move, copy semantics/home/igor > === Topic ===

move semantics and rvalue references in C++11

=== Explanation ===

Move semantics in C++11 allows you to efficiently "steal" resources from temporary objects (rvalue references, marked with &&) instead of making expensive copies, using std::move() to transfer ownership of data like moving contents between containers. Think of it like moving houses: instead of laboriously copying every item to your new home (copy semantics), you just pick up your furniture and relocate it (move semantics) - much faster and less wasteful.

=== Keywords ===

Move semantics, rvalue references, std::move, copy semantics

/home/igor > Sent to mshell (3661 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 12 Async Parallel 3 Models + Await Barrier + Synthesis

Concept: Three LLM models are launched asynchronously in parallel, each with a different instructional angle on the same C++-generated question. An `await=` barrier blocks until all three finish. A fourth LLM synthesises the three perspectives, and a C++ block renders the final output.

Use case: High-quality C++ concept explanation \u2014 collect a beginner explanation, memory-model deep dive, and a real-world analogy simultaneously, then synthesise the

best of all three. Async execution reduces total wall-clock time to the slowest model.

Key concept: `async` on an LLM directive launches it as a background process.

`cpp await=ans1,ans2,ans3` blocks until all named variables are written. The synthesising LLM receives all three via `<ans1 <ans2 <ans3`.

Flow Diagram

```
cpp >question
@1 <question  @2 <question   @3 <question
>ans1 async   >ans2 async    >ans3 async
cpp await=ans1,ans2,ans3
@1 <ans1 <ans2 <ans3 >final  (synthesise)
cpp <final                (render result)
```

Code

```cpp >question
#include <iostream>
int main() {
   std::cout <<
      "What is std::move() in C++11 and why does it matter for performance?"
      << std::endl;
   return 0;
}
```

<!--@1 <question >ans1 async
Explain the concept in the input in one sentence aimed at a C++ beginner.
Use simple language \u2014 no jargon beyond what a first-year student would know.
-->

<!--@2 <question >ans2 async

Explain the concept in the input in one sentence, focusing on the memory model:

what actually happens at the hardware / object level.

-->

<!--@3 <question >ans3 async

Explain the concept in the input in one sentence using a vivid real-world analogy

that requires no programming knowledge to understand.

-->

```cpp await=ans1,ans2,ans3

// await barrier: wait for all three async LLM jobs

#include <iostream>

int main() {

  std::cout << "[await] all three LLM responses received\n";

  return 0;

}

```

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

You have three explanations of the same C++ concept:

[ans1] is a beginner explanation,

[ans2] is a memory-model explanation,

[ans3] is a real-world analogy.

Synthesise them into one perfect sentence that is accurate, vivid, and accessible

to developers of all levels. Reply with ONLY that single sentence.

-->

```cpp <final

#include <iostream>

#include <fstream>
```

```
#include <string>
std::string slurp(const char* key) {
   std::ifstream f(std::getenv(key));
   return {std::istreambuf_iterator<char>(f), {}};
}
int main() {
   auto a1  = slurp("MSH_VAR_ans1");
   auto a2  = slurp("MSH_VAR_ans2");
   auto a3  = slurp("MSH_VAR_ans3");
   auto fin = slurp("MSH_VAR_final");
   std::string sep(60, '-');
   std::cout << "[Beginner]     " << a1
             << sep << "\n"
             << "[Memory model] " << a2
             << sep << "\n"
             << "[Analogy]      " << a3
             << sep << "\n"
             << ">>> Synthesis: " << fin << "\n";
   return 0;
}
```

Key Patterns

- `async` jobs run in parallel background processes \u2014 total time \u2248 slowest model, not the
  sum of all three.
- `await=ans1,ans2,ans3` must list every async output variable before proceeding.
- When the synthesising LLM receives multiple `<invar`, mshell injects each with a

`[varname]:` label \u2014 reference them explicitly in the prompt for clarity.

- The `cpp await=` block must still compile and run; an empty `main()` that prints a status message is the recommended pattern.

What is std::move() in C++11 and why does it matter for performance?

[async llm] Launched PID 320610 \u2192 var=ans1 (model @1)

[async llm] Launched PID 320611 \u2192 var=ans2 (model @2)

[async llm] Launched PID 320613 \u2192 var=ans3 (model @3)

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

[async] Waiting for PID 320610 (var=ans1)...

[async] PID 320610 done (var=ans1)

`std::move()` is a C++ function that tells the compiler "I'm done with this variable, so you can steal its contents instead of making an expensive copy," which makes programs much faster when working with large objects like strings or containers.

`std::move` is like giving someone the keys to your car instead of buying them an identical one\u2014it tells the compiler to transfer ownership of an object's resources (memory, handles) by copying just a few pointers rather than duplicating expensive contents, leaving the original in a safe but empty state.

/home/igor > [Beginner] `std::move()` is a C++ function that tells the compiler "I'm done with this variable, so you can steal its contents instead of making an expensive copy," which makes programs much faster when working with large objects like strings or containers.

[Memory model] `std::move` is a cast that tells the compiler an object may be safely treated as a disposable source, allowing its resources (like heap pointers or file handles) to be *rebound* to another object instead of duplicated, so at the hardware level only a few pointers or small fields are copied and the original object is left in a valid-but-empty state, avoiding expensive deep copies of underlying memory.----------

[Analogy] Handing over the deed to your house instead of making a photocopy allows the new owner to immediately occupy the physical property without the costly duplication of bricks and mortar.----------

>>> Synthesis: `std::move` is like giving someone the keys to your car instead of buying them an identical one\u2014it tells the compiler to transfer ownership of an object's

resources (memory, handles) by copying just a few pointers rather than duplicating expensive contents, leaving the original in a safe but empty state.

/home/igor > Sent to mshell (3372 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 13 WHILE Loop: Iterative Counter with LLM Commentary

Concept: A C++ block initialises a counter and a status flag. The WHILE loop runs while `status == running`. Each iteration a C++ block increments the counter and writes both values directly to their `$MSH_VAR_*` files. When the counter reaches the target the status is set to `done`, stopping the loop. An LLM generates one interesting fact about the current count. A C++ block prints the iteration result.

Use case: Iterative computation log \u2014 count through a sequence, annotate each step with LLM commentary, stop at a threshold.

Key concept: A C++ block with multiple `>outvar` must write to each `$MSH_VAR_*` file directly via `std::ofstream` \u2014 mshell does not capture stdout when multiple output variables are declared.

Flow Diagram

```
cpp >status="running"
cpp >counter="0"
 [WHILE status:running]
  cpp <counter <status >counter >status   (increment; write via MSH_VAR_*)
  @1 <counter >comment                (LLM: one interesting fact)
  cpp <comment                    (print iteration result)
```

[END_WHILE]

cpp <counter (final output)

```

## Code

```cpp >status
#include <iostream>
int main() { std::cout << "running\n"; return 0; }
```

```cpp >counter
#include <iostream>
int main() { std::cout << "0\n"; return 0; }
```

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

```cpp <counter <status >counter >status
#include <iostream>
#include <fstream>
#include <string>
int main() {
  // read current counter
  std::ifstream fc(std::getenv("MSH_VAR_counter"));
  int val; fc >> val; fc.close();
  ++val;
  // write new counter
  std::ofstream wc(std::getenv("MSH_VAR_counter"));
  wc << val << "\n"; wc.close();
  // write new status
  std::ofstream ws(std::getenv("MSH_VAR_status"));
```
```

```
  ws << (val >= 4 ? "done" : "running") << "\n"; ws.close();
  // stdout NOT captured (multiple outvars) \u2014 this just goes to terminal log
  std::cout << "[cpp] counter=" << val << "\n";
  return 0;
}
```

<!--@1 <counter >comment
The input contains a single integer.
In one sentence, state one surprising or beautiful mathematical fact about that number.
-->

```cpp <comment
#include <iostream>
#include <fstream>
#include <string>
int main() {
  std::ifstream fc(std::getenv("MSH_VAR_counter"));
  int val; fc >> val; fc.close();
  std::ifstream fm(std::getenv("MSH_VAR_comment"));
  std::string comment((std::istreambuf_iterator<char>(fm)), {});
  std::cout << "[iter " << val << "] " << comment << "\n";
  return 0;
}
```

<!--@end_while-->

```cpp <counter
#include <iostream>
#include <fstream>
```

```
int main() {
   std::ifstream f(std::getenv("MSH_VAR_counter"));
   int val; f >> val;
   std::cout << "=== WHILE done. Final counter = " << val << " ===\n";
   return 0;
}
```

Key Patterns

- **Multiple `>outvar` on a C++ block:** use `std::ofstream` to write directly to each `std::getenv("MSH_VAR_*")` path mshell does NOT capture stdout in this case.
- WHILE reads the **last non-empty line** of the condition variable.
- Use a `running`/`done` flag as the exit condition the most reliable pattern.
- All variables (`status`, `counter`) must be initialised before the loop because WHILE checks the condition before the first iteration.

running

0

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

[cpp] counter=1

The number 1 is the only positive integer that is equal to its own factorial, making it simultaneously the foundation of all multiplication and the elegant exception that proves the rule.

[iter 1] The number 1 is the only positive integer that is equal to its own factorial, making it simultaneously the foundation of all multiplication and the elegant exception that proves the rule.

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

[cpp] counter=2

The number 2 is the only even prime number, making it both the foundation of all even integers and the unique exception that bridges the gap between the multiplicative building blocks and the additive structure of mathematics.

[iter 2] The number 2 is the only even prime number, making it both the foundation of all even integers and the unique exception that bridges the gap between the multiplicative building blocks and the additive structure of mathematics.

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

[cpp] counter=3

The number 3 is the only number that equals the sum of all positive integers less than itself (1 + 2 = 3), making it the first triangular number after 1 and a perfect example of how addition can recreate multiplication's essence.

[iter 3] The number 3 is the only number that equals the sum of all positive integers less than itself (1 + 2 = 3), making it the first triangular number after 1 and a perfect example of how addition can recreate multiplication's essence.

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

[cpp] counter=4

The number 4 is the only composite number that equals the number of letters in its own English name, creating a rare self-referential loop where mathematics and language perfectly intersect.

[iter 4] The number 4 is the only composite number that equals the number of letters in its own English name, creating a rare self-referential loop where mathematics and language perfectly intersect.

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

/home/igor > === WHILE done. Final counter = 4 ===

/home/igor > Sent to mshell (2546 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 14 FOREACH: LLM Processes Each Item in a List

Concept: A C++ block creates a newline-separated list of items. FOREACH iterates over

the list line by line. On each iteration the runtime automatically sets the iterator variable

to the current item. An LLM generates a response for each item. A C++ block prints the result.

Use case: Automated C++ concept glossary \u2014 iterate over a list of C++ features, ask the LLM for a one-sentence definition of each, print the mini-glossary.

Key concept: The list must be created with one item per line (`\n`-separated). The iterator variable is set automatically by the runtime before each iteration.

Flow Diagram

```
cpp >features          (newline-separated list of C++ features)
 [FOREACH feature in features]
  @1 <feature >definition   (LLM: one-sentence definition)
  cpp <definition           (print feature name + definition)
[END_FOREACH]
cpp                  (completion message)
```

Code

```cpp >features
#include <iostream>
int main() {
   // one feature per line \u2014 mandatory for FOREACH
   std::cout << "RAII\nstd::optional\nstd::variant\nMove semantics\nConcepts (C++20)" << std::endl;
   return 0;
}
```

<!--@foreach feature in features-->

<!--@1 <feature >definition

The input contains the name of a C++ language feature or standard-library component.

In exactly one sentence, define what it is and state its primary benefit.

-->

```cpp <definition
#include <iostream>
#include <fstream>
#include <string>
int main() {
   std::ifstream ff(std::getenv("MSH_VAR_feature"));
   std::string feature((std::istreambuf_iterator<char>(ff)), {});
   while (!feature.empty() && feature.back() == '\n') feature.pop_back();
   std::ifstream fd(std::getenv("MSH_VAR_definition"));
   std::string def((std::istreambuf_iterator<char>(fd)), {});
   std::cout << "--- " << feature << " ---\n" << def << "\n\n";
   return 0;
}
```

<!--@end_foreach-->

```cpp
#include <iostream>
int main() {
   std::cout << "=== C++ Feature Glossary complete ===\n";
   return 0;
}
```

Key Patterns

- The C++ block producing the list must output **one item per line** \u2014 space-separated

  output will not split correctly.

- `MSH_VAR_feature` is automatically populated by the FOREACH runtime before each

  C++ block runs.

- Any number of C++ and LLM blocks can appear inside a FOREACH body.

- Trim trailing `\n` from iterator variables with `while (!s.empty() && s.back() == '\n') s.pop_back();`.

RAII

std::optional

std::variant

Move semantics

Concepts (C++20)

[foreach] iter 1: feature=RAII

RAII (Resource Acquisition Is Initialization) is a C++ design principle where resources like memory, file handles, or locks are automatically acquired in a constructor and released in a destructor, ensuring that resources are always properly cleaned up even when exceptions occur.

--- RAII ---

RAII (Resource Acquisition Is Initialization) is a C++ design principle where resources like memory, file handles, or locks are automatically acquired in a constructor and released in a destructor, ensuring that resources are always properly cleaned up even when exceptions occur.

[foreach] iter 2: feature=std::optional

`std::optional` is a C++17 template class that represents a value that may or may not be present, eliminating the need for error-prone null pointers or magic sentinel values while providing type-safe ways to handle optional data.

--- std::optional ---

`std::optional` is a C++17 template class that represents a value that may or may not be present, eliminating the need for error-prone null pointers or magic sentinel values while providing type-safe ways to handle optional data.

[foreach] iter 3: feature=std::variant

`std::variant` is a C++17 type-safe union that can hold one of several specified types at any given time, providing memory-efficient polymorphism without inheritance or dynamic allocation while ensuring compile-time type safety through visitor patterns.

--- std::variant ---

`std::variant` is a C++17 type-safe union that can hold one of several specified types at any given time, providing memory-efficient polymorphism without inheritance or dynamic allocation while ensuring compile-time type safety through visitor patterns.

[foreach] iter 4: feature=Move semantics

Move semantics is a C++11 feature that allows objects to transfer ownership of their resources (like dynamically allocated memory) rather than copying them, dramatically improving performance by eliminating expensive deep copies when working with temporary objects or transferring large data structures.

--- Move semantics ---

Move semantics is a C++11 feature that allows objects to transfer ownership of their resources (like dynamically allocated memory) rather than copying them, dramatically improving performance by eliminating expensive deep copies when working with temporary objects or transferring large data structures.

[foreach] iter 5: feature=Concepts (C++20)

Concepts are a C++20 feature that allows developers to specify explicit constraints on template parameters using readable syntax, providing clear compile-time requirements that produce much better error messages and make generic code more self-documenting and easier to understand.

--- Concepts (C++20) ---

Concepts are a C++20 feature that allows developers to specify explicit constraints on template parameters using readable syntax, providing clear compile-time requirements that produce much better error messages and make generic code more self-documenting and easier to understand.

/home/igor > === C++ Feature Glossary complete ===

/home/igor > Sent to mshell (3978 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 15 TRY/CATCH: Safe Execution with Error Capture

Concept: A C++ block initialises an input. The TRY block wraps a C++ block that intentionally triggers a runtime error (divide-by-zero detected at runtime, exits with `exit(1)`). Because the block exits with a non-zero code the parser skips the rest of the TRY body and jumps to CATCH. The CATCH block prints a hardcoded error message. After the TRY/CATCH a safe fallback C++ block processes the input normally.

Use case: Robust pipeline \u2014 attempt a risky computation (e.g. parsing untrusted data, calling an external binary), catch failure gracefully, continue with a safe fallback.

Key concept:

- Exit with a non-zero code (via `exit(1)` or `return 1`) to trigger CATCH.

- CATCH block does **not** use `<errvar` \u2014 print the literal string `"try_block_failed"` directly.

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

Flow Diagram

```
cpp >input            (set up input data)
 [TRY]
  cpp <input >result    (risky computation \u2014 may exit(1))
[CATCH >error]
  cpp                 (print literal "try_block_failed" message)
[END_TRY]
cpp <input >safe_result (safe fallback computation)
cpp <safe_result       (print result)
```

Code

```cpp >input
#include <iostream>
int main() {
  // A CSV line of integers \u2014 one of which is zero (will cause division issues)
  std::cout << "100,25,0,80,50\n";
  return 0;
}
```
<!--@try-->
```cpp <input >result
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <cstdlib>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_input"));
  std::string line; std::getline(f, line);
  std::istringstream ss(line);
  std::string tok;
  std::vector<int> nums;
  while (std::getline(ss, tok, ',')) nums.push_back(std::stoi(tok));
  // intentional risky operation: divide first value by each subsequent value
  // will fail when divisor == 0
  for (int i = 1; i < (int)nums.size(); ++i) {
    if (nums[i] == 0) {
```

```
            std::cerr << "ERROR: division by zero detected at index " << i << "\n";
            std::exit(1);   // triggers CATCH
        }
        std::cout << nums[0] << " / " << nums[i] << " = " << nums[0] / nums[i] << "\n";
    }
    return 0;
}
```

<!--@catch >error-->

```cpp
#include <iostream>
int main() {
    std::cout << "=== Caught error: try_block_failed ===\n"
              << "Division pipeline aborted. Switching to safe fallback.\n";
    return 0;
}
```

<!--@end_try-->

```cpp <input >safe_result
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <numeric>
int main() {
    std::ifstream f(std::getenv("MSH_VAR_input"));
```

```
    std::string line; std::getline(f, line);
    std::istringstream ss(line);
    std::string tok;
    std::vector<int> nums;
    while (std::getline(ss, tok, ',')) nums.push_back(std::stoi(tok));
    // safe: just compute sum and count (no division)
    int sum = std::accumulate(nums.begin(), nums.end(), 0);
    std::cout << "Safe fallback: count=" << nums.size()
              << " sum=" << sum
              << " mean=" << (double)sum / nums.size()
              << "\n";
    return 0;
}
```

```cpp <safe_result
#include <iostream>
#include <fstream>
#include <string>
int main() {
    std::ifstream f(std::getenv("MSH_VAR_safe_result"));
    std::string line;
    std::cout << "=== Safe Result ===\n";
    while (std::getline(f, line)) std::cout << line << "\n";
    return 0;
}
```

Key Patterns

- `std::exit(1)` (or `return 1` from `main()`) guarantees a non-zero exit code on any error.

- CATCH block must **not** use `<error` \u2014 print the literal string `"try_block_failed"` directly.

- The first block inside TRY to fail skips all remaining TRY blocks; control jumps to CATCH.

- After `<!--@end_try-->` the pipeline continues normally regardless of whether TRY or

  CATCH ran.

100,25,0,80,50

[try] executing try block

ERROR: division by zero detected at index 2

100 / 25 = 4

[try] try block failed, executing catch block

=== Caught error: try_block_failed ===

Division pipeline aborted. Switching to safe fallback.

Safe fallback: count=5 sum=255 mean=51

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

Safe fallback: count=5 sum=255 mean=51

/home/igor > Sent to mshell (3307 bytes)

Received from GUI editor:

---

mshell C++ Edition

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

Concept: A C++ block creates a two-line dataset. SPLIT divides it by lines into

`dataset_1` and `dataset_2`. Two async LLM calls analyse each half in parallel. An

`await=` barrier synchronises them. MERGE marks the reduction point. A single LLM

synthesises both analyses. A final C++ block prints everything.

Use case: Parallel analysis of two separate benchmark result sets \u2014 split the data,

analyse each independently, merge into a unified performance report.

Key concept:

- SPLIT creates variables line-by-line: line 1 \u2192 `var_1`, line 2 \u2192 `var_2`.

- SPLIT and MERGE are **visual markers** \u2014 they execute no code.

- `async` + `await=`: total time = time of the slowest call, not the sum.

Flow Diagram

```
cpp >dataset          ("set_a values\nset_b values")
 [SPLIT dataset into 2]  dataset_1, dataset_2
  @1 <dataset_1 >analysis1  async
  @2 <dataset_2 >analysis2  async
  cpp await=analysis1,analysis2
[MERGE]
  @1 <analysis1 <analysis2 >combined   (synthesise)
cpp <combined         (print full report)
```

Code

```cpp >dataset
#include <iostream>
int main() {
   // two lines \u2192 SPLIT will create dataset_1 and dataset_2
   std::cout << "benchmark_A: 12ms 15ms 11ms 14ms 16ms 13ms\n"
             << "benchmark_B: 45ms 42ms 48ms 44ms 46ms 43ms\n";
   return 0;
}
```

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

<!--@1 <dataset_1 >analysis1 async

The input contains a series of latency measurements for one benchmark run.

In one sentence, characterise the performance: mention mean, spread, and consistency.

-->

<!--@2 <dataset_2 >analysis2 async

The input contains a series of latency measurements for one benchmark run.

In one sentence, characterise the performance: mention mean, spread, and consistency.

-->

```cpp await=analysis1,analysis2
#include <iostream>
int main() {
   std::cout << "[await] both analyses ready\n";
   return 0;
}
```

<!--@merge-->

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

The first input analyses benchmark A. The second analyses benchmark B.

Write a two-sentence comparative summary: which is faster, how large is the gap,

and what does it suggest about the workloads?

-->

```cpp <combined
#include <iostream>
#include <fstream>
#include <string>
std::string slurp(const char* key) {
```

```
    std::ifstream f(std::getenv(key));
    return {std::istreambuf_iterator<char>(f), {}};
}
int main() {
    std::cout << "=== Benchmark A ===\n" << slurp("MSH_VAR_dataset_1")
              << "Analysis A: "         << slurp("MSH_VAR_analysis1") << "\n"
              << "=== Benchmark B ===\n" << slurp("MSH_VAR_dataset_2")
              << "Analysis B: "         << slurp("MSH_VAR_analysis2") << "\n"
              << "=== Merged Report ===\n" << slurp("MSH_VAR_combined") << "\n";
    return 0;
}
```

Key Patterns

- SPLIT reads `dataset` line-by-line and writes `dataset_1` (line 1), `dataset_2` (line 2), etc.

- MERGE is a **visual-only** marker \u2014 it generates no code and writes no variables.

- An `await=` barrier **must** appear before MERGE when async jobs are in flight.

- The reduction LLM receives both analyses with `[varname]:` labels \u2014 reference them in

  the prompt for clarity.

benchmark_A: 12ms 15ms 11ms 14ms 16ms 13ms

benchmark_B: 45ms 42ms 48ms 44ms 46ms 43ms

[split] dataset_1 = benchmark_A: 12ms 15ms 11ms 14ms 16ms 13ms

[split] dataset_2 = benchmark_B: 45ms 42ms 48ms 44ms 46ms 43ms

[async llm] Launched PID 321658 var=analysis1 (model @1)

[async llm] Launched PID 321659 var=analysis2 (model @2)

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

[async] Waiting for PID 321658 (var=analysis1)...

[async] PID 321658 done (var=analysis1)

Benchmark A shows consistent performance with a mean of 13.5ms and tight clustering between 11-16ms, indicating reliable and predictable latency with minimal variance of only 5ms spread.

I see only one benchmark analysis (Benchmark A with 13.5ms mean and 11-16ms range), but no information about Benchmark B to make a comparison. Please provide the analysis for Benchmark B so I can write the requested two-sentence comparative summary.

/home/igor > === Benchmark A ===

benchmark_A: 12ms 15ms 11ms 14ms 16ms 13msAnalysis A: Benchmark A shows consistent performance with a mean of 13.5ms and tight clustering between 11-16ms, indicating reliable and predictable latency with minimal variance of only 5ms spread.

=== Benchmark B ===

benchmark_B: 45ms 42ms 48ms 44ms 46ms 43msAnalysis B:

=== Merged Report ===

I see only one benchmark analysis (Benchmark A with 13.5ms mean and 11-16ms range), but no information about Benchmark B to make a comparison. Please provide the analysis for Benchmark B so I can write the requested two-sentence comparative summary.

/home/igor > Sent to mshell (3299 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 17 CONFIG Node: Parameterized Pipeline

Concept: A CONFIG block documents the pipeline parameters. C++ blocks initialise the

actual runtime variables via `std::cout`. Two LLMs generate and post-process content

based on those parameters. A C++ block reads all variables and prints a formatted report.

Use case: Reusable documentation generator \u2014 change `topic` and `audience` in two

C++ lines to produce tailored C++ concept explanations for any subject and reader level.

Key concept:

- CONFIG is a **documentation block** \u2014 it does NOT inject variables at runtime.

Always pair with C++ blocks that `std::cout` the actual values.
- `msh_run_and_capture` prints stdout to screen AND writes to the variable \u2014 avoid duplicating output with an extra print block.

Flow Diagram

```
config (topic, audience, max_sentences \u2014 documentation only)
cpp >topic = "smart pointers in C++"
cpp >audience = "junior C++ developer"
@1 <topic <audience >explanation      (generate explanation)
@2 <explanation >keywords             (extract keywords)
cpp <explanation <keywords >report    (print formatted report)
```

Code

```config
topic=smart pointers in C++
audience=junior C++ developer
max_sentences=3
```

```cpp >topic
#include <iostream>
int main() {
    std::cout << "smart pointers in C++\n";
    return 0;
}
```

```cpp >audience
```

```
#include <iostream>
int main() {
   std::cout << "junior C++ developer\n";
   return 0;
}
```

<!--@1 <topic <audience >explanation
The first input is a C++ topic to explain. The second input is the target audience.
Explain the topic for that audience in exactly 3 sentences.
Use concrete examples where possible. Avoid jargon not suitable for the audience level.
-->

<!--@2 <explanation >keywords
The input is a short technical explanation.
Extract exactly 5 keywords or key phrases as a comma-separated list.
Reply with ONLY the list \u2014 no explanation, no numbering.
-->

```cpp <explanation <keywords >report
#include <iostream>
#include <fstream>
#include <string>
std::string slurp(const char* key) {
   std::ifstream f(std::getenv(key));
   return {std::istreambuf_iterator<char>(f), {}};
}
int main() {
   auto topic      = slurp("MSH_VAR_topic");
   auto audience   = slurp("MSH_VAR_audience");
```

```
    auto explanation = slurp("MSH_VAR_explanation");
    auto keywords    = slurp("MSH_VAR_keywords");
    // trim trailing newlines for inline display
    auto trim = [](std::string s) {
        while (!s.empty() && s.back() == '\n') s.pop_back();
        return s;
    };
    std::string sep(60, '=');
    std::cout << sep << "\n"
              << "TOPIC    : " << trim(topic)    << "\n"
              << "AUDIENCE : " << trim(audience) << "\n"
              << sep << "\n"
              << "EXPLANATION:\n" << explanation
              << "KEYWORDS : "   << trim(keywords) << "\n"
              << sep << "\n";
    return 0;
}
```

Key Patterns

- The `config` block is **read-only documentation** \u2014 never rely on it to inject runtime values.

- Each config parameter needs a matching `cpp >varname` block that `std::cout`s the value.

- To create a reusable template, change only the C++ `std::cout` strings \u2014 the rest of the

  pipeline is unchanged.

[config] topic=smart pointers in C++

[config] audience=junior C++ developer

[config] max_sentences=3

smart pointers in C++

junior C++ developer

Smart pointers are C++ classes that automatically manage memory for you, so you don't have to remember to call `delete` and risk memory leaks or crashes. The most common one is `std::unique_ptr`, which acts like a regular pointer but automatically frees the memory when it goes out of scope example, `std::unique_ptr<int> ptr = std::make_unique<int>(42);` creates an integer on the heap that gets cleaned up automatically. When you need to share ownership of the same object between multiple parts of your code, use `std::shared_ptr`, which keeps track of how many pointers reference the object and only deletes it when the last one is destroyed.

smart pointers, std::unique_ptr, std::shared_ptr, automatic memory management, shared ownership

==========

TOPIC : smart pointers in C++

AUDIENCE : junior C++ developer

==========

EXPLANATION:

Smart pointers are C++ classes that automatically manage memory for you, so you don't have to remember to call `delete` and risk memory leaks or crashes. The most common one is `std::unique_ptr`, which acts like a regular pointer but automatically frees the memory when it goes out of scope for example, `std::unique_ptr<int> ptr = std::make_unique<int>(42);` creates an integer on the heap that gets cleaned up automatically. When you need to share ownership of the same object between multiple parts of your code, use `std::shared_ptr`, which keeps track of how many pointers reference the object and only deletes it when the last one is destroyed.KEYWORDS : smart pointers, std::unique_ptr, std::shared_ptr, automatic memory management, shared ownership

==========

/home/igor > Sent to mshell (3240 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 18 FOREACH + Async LLM: Parallel Batch Processing

Concept: A C++ block creates a newline-separated list of items. FOREACH iterates

line by line. On each iteration two async LLM calls run simultaneously \u2014 one generates an

explanation, the other a real-world analogy. An `await=` barrier synchronises them before

a C++ block prints the combined result.

Use case: Automated C++ algorithm cheat-sheet \u2014 for each algorithm name, produce a

plain-English explanation and a real-world analogy in parallel, then display them together.

Key concept:

- `async` inside FOREACH: for each item both LLMs run simultaneously \u2014 total time per

  item = time of the slowest model.

- Each async call must have exactly one `>outvar` \u2014 otherwise `await=` cannot match the job.

- The `await=` barrier lists all async output variables from the current iteration, comma-separated.

Flow Diagram

```
cpp >algorithms        ("QuickSort\nBinary Search\nDijkstra's Algorithm")
 [FOREACH algo in algorithms]
  @1 <algo >explanation  async
  @2 <algo >analogy      async
  cpp await=explanation,analogy
  cpp <explanation <analogy      (print pair)
[END_FOREACH]
cpp                    (completion message)
```

Code

```cpp >algorithms
```

```
#include <iostream>
int main() {
   // one algorithm per line \u2014 mandatory for FOREACH line-by-line iteration
   std::cout << "QuickSort\nBinary Search\nDijkstra's Algorithm\n";
   return 0;
}
```

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

<!--@1 <algo >explanation async

The input contains the name of a well-known algorithm.

In one sentence, explain what it does and when you would use it. Target: C++ developer.

-->

<!--@2 <algo >analogy async

The input contains the name of a well-known algorithm.

In one sentence, give a vivid real-world analogy that explains the core idea to a non-programmer.

-->

```cpp await=explanation,analogy
#include <iostream>
int main() { return 0; }
```

```cpp <explanation <analogy
#include <iostream>
#include <fstream>
#include <string>
std::string slurp(const char* key) {
   std::ifstream f(std::getenv(key));
```

```
    return {std::istreambuf_iterator<char>(f), {}};
}
int main() {
    std::ifstream fa(std::getenv("MSH_VAR_algo"));
    std::string algo((std::istreambuf_iterator<char>(fa)), {});
    while (!algo.empty() && algo.back() == '\n') algo.pop_back();
    std::cout << "=== " << algo << " ===\n"
              << "Explanation : " << slurp("MSH_VAR_explanation")
              << "Analogy     : " << slurp("MSH_VAR_analogy") << "\n";
    return 0;
}
```

<!--@end_foreach-->

```cpp
#include <iostream>
int main() {
    std::cout << "=== Algorithm Cheat-Sheet complete ===\n";
    return 0;
}
```

Key Patterns

- `async` jobs are scoped to the current FOREACH iteration \u2014 `await=` only needs to list
  the variables written in that iteration.
- The `await=` C++ block can have an empty `main()` \u2014 its only purpose is to serve as the synchronisation barrier.
- Output variables (`explanation`, `analogy`) are **overwritten** on each iteration, so the
  print block must run before the next iteration begins (which it does, because `await=`

guarantees it).

QuickSort

Binary Search

Dijkstra's Algorithm

[foreach] iter 1: algo=QuickSort

[async llm] Launched PID 321857 \u2192 var=explanation (model @1)

[async llm] Launched PID 321858 \u2192 var=analogy (model @2)

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

[async] Waiting for PID 321857 (var=explanation)...

[async] PID 321857 done (var=explanation)

QuickSort is a divide-and-conquer sorting algorithm that recursively partitions an array around a pivot element, achieving O(n log n) average performance and is ideal when you need fast in-place sorting and can tolerate $O(n^2)$ worst-case behavior, making it a good choice for `std::sort` implementations and general-purpose sorting tasks.

=== QuickSort ===

Explanation : QuickSort is a divide-and-conquer sorting algorithm that recursively partitions an array around a pivot element, achieving O(n log n) average performance and is ideal when you need fast in-place sorting and can tolerate $O(n^2)$ worst-case behavior, making it a good choice for `std::sort` implementations and general-purpose sorting tasks.

Analogy : Imagine organizing a messy stack of papers by first picking one paper as a reference, then making two piles\u2014one with papers that should go before it and one with papers that should go after it\u2014and repeating this same \u201csplit into two piles around a reference paper\u201d process on each pile until everything is perfectly ordered.

[foreach] iter 2: algo=Binary Search

[async llm] Launched PID 321904 \u2192 var=explanation (model @1)

[async llm] Launched PID 321905 \u2192 var=analogy (model @2)

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

[async] Waiting for PID 321904 (var=explanation)...

[async] PID 321904 done (var=explanation)

Binary Search is an O(log n) algorithm that finds a target value in a sorted array by repeatedly halving the search space, making it ideal for quickly locating elements in large sorted containers like `std::vector` or when implementing efficient lookups in sorted data structures.

=== Binary Search ===

Explanation : Binary Search is an O(log n) algorithm that finds a target value in a sorted array by repeatedly halving the search space, making it ideal for quickly locating elements in large sorted containers like `std::vector` or when implementing efficient lookups in sorted data structures.

Analogy : It\u2019s like searching for a word in a dictionary by opening near the middle, deciding whether your word would be earlier or later in alphabetical order, then repeatedly halving the remaining pages instead of flipping through every page one by one.

[foreach] iter 3: algo=Dijkstra's Algorithm

[async llm] Launched PID 321950 \u2192 var=explanation (model @1)

[async llm] Launched PID 321951 \u2192 var=analogy (model @2)

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

[async] Waiting for PID 321950 (var=explanation)...

[async] PID 321950 done (var=explanation)

Dijkstra's Algorithm finds the shortest path from a starting vertex to all other vertices in a weighted graph with non-negative edge weights, making it ideal for routing problems like GPS navigation, network packet routing, or game AI pathfinding where you need optimal routes through connected nodes.

=== Dijkstra's Algorithm ===

Explanation : Dijkstra's Algorithm finds the shortest path from a starting vertex to all other vertices in a weighted graph with non-negative edge weights, making it ideal for routing problems like GPS navigation, network packet routing, or game AI pathfinding where you need optimal routes through connected nodes.

Analogy : Imagine planning the quickest delivery route from one store to every house in a city by always expanding from the locations you can currently reach the fastest, updating any house\u2019s planned route whenever you discover a shorter path through roads you\u2019ve already evaluated.

/home/igor > === Algorithm Cheat-Sheet complete ===

/home/igor > Sent to mshell (5018 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 19 WHILE Quality Gate: Generate Until Threshold

Concept: Initialises a task, a `running/done` status flag, an iteration counter, a score, and an empty result. The WHILE loop runs while `status == running`. Each iteration: a C++ block increments the counter (writing directly to `$MSH_VAR_*`), LLM @1 generates a C++ code snippet, LLM @2 rates it 1\u201310 returning only the integer, a C++ block reads the score, prints a log line, and writes `done` to status when `score >= 8`.

Use case: Automated code quality gate \u2014 keep regenerating a C++ snippet until a scorer model judges it good enough.

Key concept:

- Scorer (@2) must return a bare integer \u2014 prompt with "Reply with ONLY the integer."
- Strip whitespace from the score string before numeric comparison in C++.
- The status-update C++ block must always write to `>status` \u2014 even when continuing.
- All variables must be initialised before the loop (WHILE checks condition first).

Flow Diagram

```
cpp >task  >status="running"  >iteration="0"  >score="0"  >snippet=""
 [WHILE status:running]
  cpp <iteration >iteration        (counter++ via MSH_VAR_*)
  @1 <task >snippet               (generate C++ snippet)
  @2 <snippet >score              (rate 1-10, integer only)
  cpp <iteration <score <snippet >status   (log + threshold check)
```

[END_WHILE]

cpp <snippet <score (print accepted snippet)

```

## Code

```cpp >task
#include <iostream>
int main() {
  std::cout << "Write a modern C++17 function template that safely clamps a value "
            "between a minimum and maximum, using std::clamp or equivalent. "
            "Must be generic, constexpr, and include a usage example in main().";
  return 0;
}
```

```cpp >status
#include <iostream>
int main() { std::cout << "running\n"; return 0; }
```

```cpp >iteration
#include <iostream>
int main() { std::cout << "0\n"; return 0; }
```

```cpp >score
#include <iostream>
int main() { std::cout << "0\n"; return 0; }
```

```cpp >snippet

```
#include <iostream>
int main() { std::cout << "\n"; return 0; }
```
<!--@while status:running-->
```cpp <iteration >iteration
#include <iostream>
#include <fstream>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_iteration"));
  int val; f >> val; f.close();
  ++val;
  std::ofstream w(std::getenv("MSH_VAR_iteration"));
  w << val << "\n"; w.close();
  // stdout must be only the number \u2014 mshell captures it as the new variable value
  std::cout << val << "\n";
  return 0;
}
```
<!--@1 <task >snippet
The input describes a C++ programming task.
Return ONLY valid C++ source code \u2014 no markdown fences, no explanation.
Must compile with: g++ -std=c++17 -O2
-->
<!--@2 <snippet >score
The input is C++ source code.
Rate it 1-10 on these criteria: correctness, use of modern C++17 features, readability.
Reply with ONLY the integer score \u2014 nothing else, no explanation.

-->

```cpp <iteration <score <snippet >status
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <algorithm>
int main() {
  auto slurp = [](const char* key) -> std::string {
    std::ifstream f(std::getenv(key));
    return {std::istreambuf_iterator<char>(f), {}};
  };
  std::string iter_s  = slurp("MSH_VAR_iteration");
  std::string score_s = slurp("MSH_VAR_score");
  // strip whitespace from score
  score_s.erase(std::remove_if(score_s.begin(), score_s.end(),
            [](char c){ return std::isspace((unsigned char)c); }),
            score_s.end());
  int score = 0;
  try { score = std::stoi(score_s); } catch(...) { score = 0; }
  int iter = 0;
  try { iter = std::stoi(iter_s); } catch(...) {}
  std::cout << "[Iter " << iter << "] Score=" << score << "\n";
  std::string new_status = (score >= 8) ? "done" : "running";
  std::ofstream ws(std::getenv("MSH_VAR_status"));
  ws << new_status << "\n"; ws.close();
  std::cout << "[status -> " << new_status << "]\n";
```

```
    return 0;
}
```
<!--@end_while-->
```cpp <snippet <score
#include <iostream>
#include <fstream>
#include <string>
int main() {
    std::ifstream fs(std::getenv("MSH_VAR_score"));
    std::string score_s((std::istreambuf_iterator<char>(fs)), {});
    std::ifstream fsnip(std::getenv("MSH_VAR_snippet"));
    std::string snippet((std::istreambuf_iterator<char>(fsnip)), {});
    std::cout << "=== Accepted Snippet (score=" << score_s << ") ===\n"
              << snippet << "\n";
    return 0;
}
```
Key Patterns
- Write to `>status` using `std::ofstream` inside the same C++ block that checks the threshold both `>score` and `>status` are written in that one block.
- `std::remove_if` + `std::isspace` is the C++ equivalent of `tr -d '[:space:]'`.
- Always initialise all loop variables before the `<!--@while-->` comment.
- The status-write C++ block must always produce a value for `>status`, even when continuing never leave it unwritten.

Write a modern C++17 function template that safely clamps a value between a minimum and maximum, using std::clamp or equivalent. Must be generic, constexpr, and include a usage example in main().running

0

0

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

1

```
#include <iostream>
#include <algorithm>
template<typename T>
constexpr T safe_clamp(const T& value, const T& min_val, const T& max_val) {
    return std::clamp(value, min_val, max_val);
}
int main() {
    constexpr int result1 = safe_clamp(15, 0, 10);
    constexpr double result2 = safe_clamp(2.5, 1.0, 5.0);
    constexpr int result3 = safe_clamp(-5, 0, 10);
    std::cout << result1 << std::endl;
    std::cout << result2 << std::endl;
    std::cout << result3 << std::endl;
    return 0;
}
```

9

[Iter 1] Score=9

[status -> done]

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

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

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

```
#include <iostream>
#include <algorithm>
```

```
template<typename T>
constexpr T safe_clamp(const T& value, const T& min_val, const T& max_val) {
    return std::clamp(value, min_val, max_val);
}
int main() {
    constexpr int result1 = safe_clamp(15, 0, 10);
    constexpr double result2 = safe_clamp(2.5, 1.0, 5.0);
    constexpr int result3 = safe_clamp(-5, 0, 10);
    std::cout << result1 << std::endl;
    std::cout << result2 << std::endl;
    std::cout << result3 << std::endl;
    return 0;
}
```

/home/igor > Sent to mshell (5535 bytes)

Received from GUI editor:

---

mshell C++ Edition

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

Concept: A C++ block stores a multi-sentence text. Three further C++ blocks split the text into chunks (sent1, sent2, sent3). Three async LLM @1 calls process each chunk in parallel extracting the main concept in three words. An `await=` barrier waits for all three. The MERGE node marks the reduce point. LLM @2 synthesises the three labels into one coherent theme. A final C++ block prints the map results and the reduced result.

Use case: Parallel concept extraction from a long C++ design document split the document into sections, extract the core idea from each section simultaneously, then synthesise an overall architectural theme.

Key concept:

- Map phase: N parallel LLMs, each on its own chunk total time = slowest chunk.
- Reduce phase: one LLM receives all results via multiple `<invar`; mshell injects `[varname]:` labels for each.
- AWAIT before MERGE is mandatory MERGE is a visual marker only.

Flow Diagram

```
cpp >raw_text
cpp <raw_text >sent1 / cpp <raw_text >sent2 / cpp <raw_text >sent3  (split)
 @1 <sent1 >analysis1  async
 @1 <sent2 >analysis2  async MAP
 @1 <sent3 >analysis3  async
 cpp await=analysis1,analysis2,analysis3
[MERGE]
 @2 <analysis1 <analysis2 <analysis3 >summary   (REDUCE)
 cpp <analysis1 <analysis2 <analysis3 <summary    (print)
```

Code

```cpp >raw_text
#include <iostream>
int main() {
   std::cout <<
      "C++ templates enable compile-time polymorphism and zero-cost abstractions. "
      "The RAII idiom ties resource lifetimes to object scopes, eliminating leaks. "
```

```
    "Move semantics transfer ownership of resources without copying, maximising throughput. "
    "Concepts constrain template parameters, providing clear error messages. "
    "Coroutines allow cooperative multitasking with minimal overhead."
    << std::endl;
  return 0;
}
```

```cpp <raw_text >sent1
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_raw_text"));
  std::string text((std::istreambuf_iterator<char>(f)), {});
  std::vector<std::string> sents;
  std::string s;
  for (char c : text) {
    s += c;
    if (c == '.') { sents.push_back(s); s.clear(); }
  }
  if (sents.size() >= 1) std::cout << sents[0] << "\n";
  return 0;
}
```

```cpp <raw_text >sent2
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
int main() {
    std::ifstream f(std::getenv("MSH_VAR_raw_text"));
    std::string text((std::istreambuf_iterator<char>(f)), {});
    std::vector<std::string> sents;
    std::string s;
    for (char c : text) {
        s += c;
        if (c == '.') { sents.push_back(s); s.clear(); }
    }
    // sentences 1 and 2 (0-indexed)
    std::string out;
    if (sents.size() >= 2) out += sents[1];
    if (sents.size() >= 3) out += " " + sents[2];
    std::cout << out << "\n";
    return 0;
}
```

```cpp <raw_text >sent3
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
```

```
int main() {
   std::ifstream f(std::getenv("MSH_VAR_raw_text"));
   std::string text((std::istreambuf_iterator<char>(f)), {});
   std::vector<std::string> sents;
   std::string s;
   for (char c : text) {
      s += c;
      if (c == '.') { sents.push_back(s); s.clear(); }
   }
   std::string out;
   if (sents.size() >= 4) out += sents[3];
   if (sents.size() >= 5) out += " " + sents[4];
   std::cout << out << "\n";
   return 0;
}
```

<!--@1 <sent1 >analysis1 async
The input contains one or two sentences of C++ technical text.
State the main C++ design concept described in 3 words maximum.
-->
<!--@1 <sent2 >analysis2 async
The input contains one or two sentences of C++ technical text.
State the main C++ design concept described in 3 words maximum.
-->
<!--@1 <sent3 >analysis3 async
The input contains one or two sentences of C++ technical text.
State the main C++ design concept described in 3 words maximum.

-->

```cpp await=analysis1,analysis2,analysis3
#include <iostream>
int main() {
   std::cout << "[await] all three map jobs complete\n";
   return 0;
}
```

<!--@merge-->

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

The input contains three short C++ concept labels extracted from different parts of a document.

Synthesise them into one coherent theme sentence describing the overarching design philosophy.

-->

```cpp <analysis1 <analysis2 <analysis3 <summary
#include <iostream>
#include <fstream>
#include <string>
std::string slurp(const char* key) {
   std::ifstream f(std::getenv(key));
   return {std::istreambuf_iterator<char>(f), {}};
}
int main() {
   std::cout << "=== Map Results ===\n"
             << "Chunk 1: " << slurp("MSH_VAR_analysis1")
             << "Chunk 2: " << slurp("MSH_VAR_analysis2")
             << "Chunk 3: " << slurp("MSH_VAR_analysis3")
```

```
        << "\n=== Reduce / Theme ===\n"
        << slurp("MSH_VAR_summary") << "\n";
    return 0;
}
```

Key Patterns

- Three independent C++ blocks each parse `raw_text` to extract different sentence ranges
  clean, compiled, no shell scripting required.
- `await=analysis1,analysis2,analysis3` must list **all** async output variables before MERGE.
- The reduction LLM (@2) receives `[analysis1]:`, `[analysis2]:`, `[analysis3]:` labels
  automatically when multiple `<invar` are provided.

C++ templates enable compile-time polymorphism and zero-cost abstractions. The RAII idiom ties resource lifetimes to object scopes, eliminating leaks. Move semantics transfer ownership of resources without copying, maximising throughput. Concepts constrain template parameters, providing clear error messages. Coroutines allow cooperative multitasking with minimal overhead.

C++ templates enable compile-time polymorphism and zero-cost abstractions.

The RAII idiom ties resource lifetimes to object scopes, eliminating leaks. Move semantics transfer ownership of resources without copying, maximising throughput.

Concepts constrain template parameters, providing clear error messages. Coroutines allow cooperative multitasking with minimal overhead.

[async llm] Launched PID 322470 var=analysis1 (model @1)

[async llm] Launched PID 322471 var=analysis2 (model @1)

[async llm] Launched PID 322473 var=analysis3 (model @1)

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

[async] Waiting for PID 322470 (var=analysis1)...

[async] PID 322470 done (var=analysis1)

Template metaprogramming

Emphasize robust template metaprogramming with clear constraints while optimizing resource management

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

Chunk 1: Template metaprogramming

Chunk 2: Resource management optimizationChunk 3: Template constraints

=== Reduce / Theme ===

Emphasize robust template metaprogramming with clear constraints while optimizing resource management

---

/home/igor > Sent to mshell (4817 bytes)

Received from GUI editor:

---

mshell C++ Edition

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

Concept: Initialises a task, `result=fail`, and `last_error=none`. A LOOP runs up to 3

times, exiting early when `result == ok`. Each iteration: LLM @1 generates C++ code

(seeing the previous error or "none"). A C++ block prints the generated code. The TRY

block compiles and runs it \u2014 writing "ok" on success. CATCH writes "fail" to result without

reading the catch variable. On the next iteration LLM @1 sees `last_error` and

self-corrects.

Use case: Automated self-healing C++ code generation \u2014 if the generated code fails to

compile or crashes, retry with error context until it works.

Key concept:

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

- `last_error` is initialised to `"none"` before the loop.

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

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

Flow Diagram

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

Code

```cpp >task
#include <iostream>
int main() {
    std::cout <<
        "Write a C++ program that reads a JSON-like string "
        "'{\"name\": \"Alice\", \"age\": 30}' hardcoded as a std::string, "
        "manually parses the 'name' value (no external libraries), "
        "and prints it. Must compile with g++ -std=c++17 -O2."
        << std::endl;
    return 0;
```

}

```

```cpp >result

#include <iostream>

int main() { std::cout << "fail\n"; return 0; }

```

```cpp >last_error

#include <iostream>

int main() { std::cout << "none\n"; return 0; }

```

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

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

The first input describes a C++ programming task.

The second input contains the error from the previous attempt (or "none" if this is the first).

Return ONLY valid C++ source code \u2014 no fences, no explanation.

Must compile with: g++ -std=c++17 -O2

If there was a previous error, fix it.

-->

```cpp <code

#include <iostream>

#include <fstream>

#include <string>

int main() {

  std::cout << "=== Generated Code ===\n";

  std::ifstream f(std::getenv("MSH_VAR_code"));

  std::string line;

```
    while (std::getline(f, line)) std::cout << line << "\n";
    return 0;
}
```

<!--@try-->

```cpp <code
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
int main() {
    const char* code_path = std::getenv("MSH_VAR_code");
    // copy to stable .cpp path before compiling
    std::string cp_cmd = std::string("cp -- \"") + code_path + "\" /tmp/msh_p21_gen.cpp";
    std::system(cp_cmd.c_str());
    // compile
    if (std::system(
            "g++ -std=c++17 -O2 -o /tmp/msh_p21_bin /tmp/msh_p21_gen.cpp
2>/tmp/msh_p21_err.txt"
        ) != 0) {
        std::ifstream ef("/tmp/msh_p21_err.txt");
        std::string err((std::istreambuf_iterator<char>(ef)), {});
        std::cerr << err;
        std::exit(1);
    }
    // run \u2014 stdout goes to terminal only (no >outvar on this block)
    std::system("/tmp/msh_p21_bin");
```

```
    // write ok to result file directly
    std::ofstream wr(std::getenv("MSH_VAR_result"));
    wr << "ok\n"; wr.close();
    // print ok as last stdout line \u2014 LOOP checks last stdout line
    std::cout << "ok\n";
    return 0;
}
```
<!--@catch >last_error-->
```cpp >result
#include <iostream>
#include <fstream>
#include <string>
int main() {
    std::cout << "=== Error on this attempt: try_block_failed ===\n";
    // write fail to result
    std::ofstream wr(std::getenv("MSH_VAR_result"));
    wr << "fail\n"; wr.close();
    std::cout << "fail\n";
    return 0;
}
```
<!--@end_try-->
<!--@end_loop-->
```cpp <result
#include <iostream>
#include <fstream>
```

```
#include <string>
int main() {
   std::ifstream f(std::getenv("MSH_VAR_result"));
   std::string r((std::istreambuf_iterator<char>(f)), {});
   std::cout << "=== Final status: " << r << " ===\n";
   return 0;
}
```

Key Patterns

- TRY fires CATCH when the C++ block exits with a non-zero code (`std::exit(1)`).
- CATCH declares `>last_error` so the parser registers it, but the block does **not** read it
  write `fail` to `>result` directly with `std::ofstream`.
- The compile error is sent to stderr so mshell logs it, but the CATCH mechanism is
  triggered solely by the non-zero exit code.
- `LOOP reads the last non-empty line` \u2014 always write `ok` or `fail` as the only line.

Write a C++ program that reads a JSON-like string '{"name": "Alice", "age": 30}' hardcoded as a std::string, manually parses the 'name' value (no external libraries), and prints it. Must compile with g++ -std=c++17 -O2.

fail

none

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

```
#include <iostream>
#include <string>
int main() {
   std::string json = "{\"name\": \"Alice\", \"age\": 30}";
   std::size_t name_pos = json.find("\"name\"");
   if (name_pos != std::string::npos) {
      std::size_t colon_pos = json.find(":", name_pos);
```

```
    if (colon_pos != std::string::npos) {
      std::size_t first_quote = json.find("\"", colon_pos);
      if (first_quote != std::string::npos) {
        std::size_t second_quote = json.find("\"", first_quote + 1);
        if (second_quote != std::string::npos) {
          std::string name = json.substr(first_quote + 1, second_quote - first_quote - 1);
          std::cout << name << std::endl;
        }
      }
    }
  }
  return 0;
}
[try] executing try block
=== Generated Code ===
#include <iostream>
#include <string>
int main() {
  std::string json = "{\"name\": \"Alice\", \"age\": 30}";
  std::size_t name_pos = json.find("\"name\"");
  if (name_pos != std::string::npos) {
    std::size_t colon_pos = json.find(":", name_pos);
    if (colon_pos != std::string::npos) {
      std::size_t first_quote = json.find("\"", colon_pos);
      if (first_quote != std::string::npos) {
        std::size_t second_quote = json.find("\"", first_quote + 1);
        if (second_quote != std::string::npos) {
```

```
            std::string name = json.substr(first_quote + 1, second_quote - first_quote - 1);
            std::cout << name << std::endl;
        }
      }
    }
  }
  return 0;
}
```

Alice

ok

[try] try block succeeded

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

/home/igor > === Final status: ok

/home/igor > Sent to mshell (4425 bytes)

Received from GUI editor:

---

mshell C++ Edition

Pattern 22 Multi-Variable Output: Structured Field Extraction

Concept: A C++ block stores a technical description as input. LLM @1 responds in a strict three-line format. A C++ block with three output variables parses the response with `std::regex` and writes each field directly to the corresponding `$MSH_VAR_*` file. LLM @2 rewrites the summary for the detected audience level.

Use case: Structured metadata extraction from C++ library documentation split a raw description into summary, key types, and audience level; then adapt the summary for that audience.

Key concept:

- Multiple `>outvar` on a C++ block: mshell runs the block without capturing stdout
  the block MUST write to `$MSH_VAR_*` files directly via `std::ofstream`.
- Multiple `>outvar` on an LLM directive: the FULL response is copied identically into
  each variable it does NOT split the response.
- `MSH_VAR_*` environment variables are pre-set by the parser before running the block.

Flow Diagram

```
cpp >input
@1 <input >raw_response            (LLM: strict 3-line format)
cpp <raw_response >summary >types >audience
  (parse with std::regex; write fields via std::ofstream to MSH_VAR_*)
@2 <summary <audience >adaptation   (LLM: rewrite for audience)
cpp <adaptation                    (print)
```

Code

```cpp >input
#include <iostream>
int main() {
   std::cout <<
      "std::unordered_map is a hash-table-based associative container introduced "
      "in C++11 that provides average O(1) lookup, insertion, and deletion. "
      "It stores key-value pairs with no guaranteed ordering, making it ideal for "
      "fast lookups when iteration order does not matter."
      << std::endl;
   return 0;
}
```

```
<!--@1 <input >raw_response
Respond in exactly this format (3 lines, no extra text, no blank lines):
SUMMARY: one sentence paraphrase of what the described component does
TYPES: up to 3 C++ type names or concepts directly involved, comma-separated
AUDIENCE: one word only \u2014 beginner / intermediate / expert
-->
```cpp <raw_response >summary >types >audience
#include <iostream>
#include <fstream>
#include <string>
#include <regex>
int main() {
  // read raw_response
  std::ifstream fr(std::getenv("MSH_VAR_raw_response"));
  std::string text((std::istreambuf_iterator<char>(fr)), {});
  // parse with regex
  auto extract = [&](const std::string& label) -> std::string {
    std::regex re(label + ":\\s*(.+)");
    std::smatch m;
    if (std::regex_search(text, m, re)) return m[1].str();
    return "n/a";
  };
  std::string summary  = extract("SUMMARY");
  std::string types    = extract("TYPES");
  std::string audience = extract("AUDIENCE");
  // write each field to its variable file directly
```
```

```
  {
    std::ofstream ws(std::getenv("MSH_VAR_summary"));
    ws << summary << "\n";
  }
  {
    std::ofstream wt(std::getenv("MSH_VAR_types"));
    wt << types << "\n";
  }
  {
    std::ofstream wa(std::getenv("MSH_VAR_audience"));
    wa << audience << "\n";
  }
  // stdout is NOT captured (multiple outvars) \u2014 goes to terminal log only
  std::cout << "Summary  : " << summary  << "\n"
            << "Types    : " << types    << "\n"
            << "Audience : " << audience << "\n";
  return 0;
}
```

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

The first input is a technical summary. The second input is the target audience level (beginner / intermediate / expert).

Rewrite the summary for that audience in one sentence.

Use vocabulary and analogies appropriate for that level.

-->

```cpp <adaptation
#include <iostream>
```

```
#include <fstream>

#include <string>

int main() {

   std::ifstream f(std::getenv("MSH_VAR_adaptation"));

   std::string a((std::istreambuf_iterator<char>(f)), {});

   std::cout << "\n=== Adapted Version ===\n" << a << "\n";

   return 0;

}
```

Key Patterns

- **Multiple `>outvar` on a C++ block:** always write directly to `std::getenv("MSH_VAR_*")`

  with `std::ofstream` \u2014 stdout is ignored by the parser in this mode.

- `std::regex` is the idiomatic C++11 way to parse structured LLM output.

- Each `std::ofstream` write should be scoped (use `{}` blocks) so the file is flushed

  and closed before the next write.

- Validate extracted fields and fall back to `"n/a"` to avoid empty variable files.

std::unordered_map is a hash-table-based associative container introduced in C++11 that provides average O(1) lookup, insertion, and deletion. It stores key-value pairs with no guaranteed ordering, making it ideal for fast lookups when iteration order does not matter.

SUMMARY: std::unordered_map is a hash table that stores key-value pairs with fast average constant-time operations but no ordering guarantees.

TYPES: hash table, key-value pairs, associative container

AUDIENCE: intermediate

Summary : std::unordered_map is a hash table that stores key-value pairs with fast average constant-time operations but no ordering guarantees.

Types   : hash table, key-value pairs, associative container

Audience : intermediate

`std::unordered_map` is a hash table that lets you quickly look up values by keys in average constant time, but it doesn\u2019t keep the elements in any particular order.

/home/igor >

=== Adapted Version ===

`std::unordered_map` is a hash table that lets you quickly look up values by keys in average constant time, but it doesn't keep the elements in any particular order.

/home/igor > Sent to mshell (5658 bytes)

Received from GUI editor:

---

mshell C++ Edition

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

Concept: A CONFIG block documents the parameters. C++ blocks initialise all runtime variables. The WHILE loop runs while `status == running`. Each iteration: a C++ block increments the counter, LLM @1 generates a C++ code explanation, LLM @2 scores it for clarity and correctness (1\u201310), a C++ block checks the score and sets `done` when `quality >= 7`. After the loop LLM @3 polishes the final explanation for publication. A C++ block prints the result with metrics.

Use case: Fully parameterised C++ teaching pipeline \u2014 automatically iterate until the

explanation of a C++ concept meets a quality bar, then polish it for documentation.

Change two C++ `std::cout` lines to switch topic and audience.

Key concept:

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

- CONFIG makes the pipeline a reusable template.

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

Flow Diagram

```
config (subject, target_audience, quality_threshold doc only)
cpp >subject  >target_audience  >status="running"  >iteration="0"
cpp >quality="0"  >explanation=""
 [WHILE status:running]
  cpp <iteration >iteration
  @1 <subject <target_audience >explanation   (generate)
  @2 <explanation <target_audience >quality   (score 1-10)
  cpp <iteration <quality >status           (log + threshold)
[END_WHILE]
@3 <explanation >final_polish              (polish)
cpp <final_polish <quality <iteration        (print with metrics)
```

Code

```config
subject=std::shared_ptr and ownership semantics
target_audience=intermediate C++ developer
quality_threshold=7
```

```cpp >subject
#include <iostream>
int main() {
   std::cout << "std::shared_ptr and ownership semantics\n";
   return 0;
}
```

```cpp >target_audience
```

```
#include <iostream>
int main() {
   std::cout << "intermediate C++ developer\n";
   return 0;
}
```

```cpp >status
#include <iostream>
int main() { std::cout << "running\n"; return 0; }
```

```cpp >iteration
#include <iostream>
int main() { std::cout << "0\n"; return 0; }
```

```cpp >quality
#include <iostream>
int main() { std::cout << "0\n"; return 0; }
```

```cpp >explanation
#include <iostream>
int main() { std::cout << "\n"; return 0; }
```

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

```cpp <iteration >iteration
#include <iostream>
#include <fstream>
int main() {
```

```
    std::ifstream f(std::getenv("MSH_VAR_iteration"));
    int val; f >> val; f.close();
    ++val;
    std::ofstream w(std::getenv("MSH_VAR_iteration"));
    w << val << "\n"; w.close();
    std::cout << val << "\n";
    return 0;
}
```

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

The first input is a C++ subject to explain. The second is the target audience.

Write exactly 3 sentences explaining the subject for that audience.

Be concrete \u2014 include a brief code example inline if it helps clarity.

-->

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

The first input is a C++ explanation. The second is the target audience it was written for.

Rate the explanation 1-10 on: technical accuracy, clarity for that audience, and conciseness.

Reply with ONLY the integer score \u2014 no explanation, no punctuation.

-->

```cpp <iteration <quality >status
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
int main() {
    auto slurp = [](const char* key) -> std::string {
        std::ifstream f(std::getenv(key));
```

```
    return {std::istreambuf_iterator<char>(f), {}};
  };
  std::string iter_s  = slurp("MSH_VAR_iteration");
  std::string score_s = slurp("MSH_VAR_quality");
  // strip whitespace
  score_s.erase(std::remove_if(score_s.begin(), score_s.end(),
            [](char c){ return std::isspace((unsigned char)c); }),
            score_s.end());
  int score = 0;
  try { score = std::stoi(score_s); } catch(...) {}
  std::cout << "[Iter " << iter_s << "] Quality: " << score << "\n";
  std::string new_status = (score >= 7) ? "done" : "running";
  std::ofstream ws(std::getenv("MSH_VAR_status"));
  ws << new_status << "\n"; ws.close();
  return 0;
}
```

<!--@end_while-->

<!--@3 <explanation >final_polish

The input is a C++ technical explanation that has passed a quality review.

Polish it lightly for final publication in developer documentation.

Keep exactly 3 sentences. No markdown formatting. No code fences.

-->

```cpp <final_polish <quality <iteration
#include <iostream>
#include <fstream>
#include <string>
```

```
std::string slurp(const char* key) {
   std::ifstream f(std::getenv(key));
   return {std::istreambuf_iterator<char>(f), {}};
}
int main() {
   auto polish  = slurp("MSH_VAR_final_polish");
   auto quality = slurp("MSH_VAR_quality");
   auto iter    = slurp("MSH_VAR_iteration");
   // trim
   auto trim = [](std::string s) {
      while (!s.empty() && s.back() == '\n') s.pop_back();
      return s;
   };
   std::string sep(60, '=');
   std::cout << sep << "\n"
             << "Final (score=" << trim(quality)
             << ", iterations=" << trim(iter) << ")\n"
             << sep << "\n"
             << polish
             << sep << "\n";
   return 0;
}
```

Key Patterns

- CONFIG parameters each need a matching `cpp >varname` block \u2014 CONFIG itself injects nothing.

- @2 receives `<target_audience` **explicitly** so the scorer knows who it's scoring for.

- Three model roles: generator (@1), scorer (@2), finisher (@3) \u2014 each has a single

  focused responsibility.

- Strip whitespace from the score string in C++ before numeric comparison.

 [config] subject=std::shared_ptr and ownership semantics

[config] target_audience=intermediate C++ developer

[config] quality_threshold=7

std::shared_ptr and ownership semantics

intermediate C++ developer

running

0

0

[while] iteration 1 condition met, executing body

1

`std::shared_ptr` implements shared ownership through reference counting, where multiple `shared_ptr` instances can safely point to the same object and automatically delete it when the last pointer is destroyed. For example, `auto ptr1 = std::make_shared<int>(42); auto ptr2 = ptr1;` creates two pointers sharing ownership of the same integer, with `ptr1.use_count()` returning 2. This solves the common problem of determining when to delete shared resources, but be aware that circular references (A points to B, B points to A) can create memory leaks that require `std::weak_ptr` to break the cycle.

10

[Iter 1

] Quality: 10

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

std::shared_ptr enforces safe shared ownership through reference counting, automatically deleting the managed object only when the last pointer is destroyed to prevent memory leaks. For instance, initializing two pointers as auto ptr1 = std::make_shared<int>(42) and auto ptr2 = ptr1 shares ownership of the same integer, with ptr1.use_count() returning 2. This resolves resource deletion timing challenges, but circular references where A points to B and B points to A can cause leaks requiring std::weak_ptr to break the cycle.

==========

Final (score=10, iterations=1)

==========

std::shared_ptr enforces safe shared ownership through reference counting, automatically deleting the managed object only when the last pointer is destroyed to prevent memory leaks. For instance, initializing two pointers as auto ptr1 = std::make_shared<int>(42) and auto ptr2 = ptr1 shares ownership of the same integer, with ptr1.use_count() returning 2. This resolves resource deletion timing challenges, but circular references where A points to B and B points to A can cause leaks requiring std::weak_ptr to break the cycle.==========

/home/igor > Sent to mshell (4292 bytes)

Received from GUI editor:

---

mshell C++ Edition

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

Concept: A C++ block creates a newline-separated list of C++ expressions (some valid, one intentionally broken). FOREACH iterates line by line. For each item the TRY block runs a C++ block that compiles and evaluates the expression. On success LLM @1 describes the result; a C++ block prints `[OK]`. On failure (non-zero exit) CATCH prints `[ERR]` as a hardcoded literal. The loop continues to the next item regardless.

Use case: Batch C++ expression evaluator with per-expression error isolation \u2014 one

malformed expression never stops the entire batch.

Key concept:

- TRY/CATCH inside FOREACH = per-item error isolation.
- CATCH block does **not** use `<parse_error` \u2014 prints the literal string directly.
- Do not create dependencies on variables written inside TRY \u2014 if TRY fails those variables are not written.

Flow Diagram

```
```

cpp >expressions   (newline-separated list; one broken)
 [FOREACH expr in expressions]
  [TRY]
   cpp <expr >result   (compile + evaluate may exit(1))
   @1 <result >insight (LLM: describe the result)
   cpp <insight   (print [OK])
  [CATCH >expr_error]
   cpp   (print [ERR] literal string)
  [END_TRY]
[END_FOREACH]
cpp   (completion message)
```

## Code

```cpp >expressions
#include <iostream>
int main() {
   // Three expressions, one line each; the second is intentionally invalid C++
   std::cout << "1 + 2 * 3\n"
             << "int x = @@@INVALID@@@\n"
             << "std::string(5, 'X')\n";
   return 0;
}
```

<!--@foreach expr in expressions-->
<!--@try-->

```cpp <expr >result
#include <iostream>
```
```

```
#include <fstream>
#include <string>
#include <cstdlib>
int main() {
  std::ifstream f(std::getenv("MSH_VAR_expr"));
  std::string expr((std::istreambuf_iterator<char>(f)), {});
  // trim
  while (!expr.empty() && (expr.back() == '\n' || expr.back() == ' ')) expr.pop_back();
  // Write a tiny C++ program that evaluates the expression and prints it
  std::string src =
    "#include <iostream>\n"
    "#include <string>\n"
    "int main() {\n"
    "    auto val = (" + expr + ");\n"
    "    std::cout << val << \"\\n\";\n"
    "    return 0;\n"
    "}\n";
  // write to temp file
  {
    std::ofstream tsrc("/tmp/msh_p24_expr.cpp");
    tsrc << src;
  }
  // compile
  int ret = std::system(
    "g++ -std=c++17 -O2 -o /tmp/msh_p24_expr_bin /tmp/msh_p24_expr.cpp
2>/tmp/msh_p24_err.txt");
  if (ret != 0) {
```

```
    std::ifstream ef("/tmp/msh_p24_err.txt");
    std::string err((std::istreambuf_iterator<char>(ef)), {});
    std::cerr << err;
    std::exit(1);
  }
  // run and capture output into result variable
  std::system("/tmp/msh_p24_expr_bin");
  return 0;
}
```

<!--@1 <result >insight

The input contains the evaluated result of a C++ expression.

In one sentence, describe what kind of value this is and what operation produced it.

-->

```cpp <insight
#include <iostream>
#include <fstream>
#include <string>
int main() {
  std::ifstream fi(std::getenv("MSH_VAR_insight"));
  std::string insight((std::istreambuf_iterator<char>(fi)), {});
  std::cout << "[OK] " << insight << "\n";
  return 0;
}
```

<!--@catch >expr_error-->

```cpp
```

```
#include <iostream>
int main() {
   std::cout << "[ERR] Expression failed: try_block_failed\n";
   return 0;
}
```
<!--@end_try-->
<!--@end_foreach-->
```cpp
#include <iostream>
int main() {
   std::cout << "=== Batch complete. Errors were isolated, pipeline never stopped. ===\n";
   return 0;
}
```
Key Patterns
- TRY/CATCH inside FOREACH = **per-item error isolation** \u2014 a broken expression never
  halts the remaining iterations.
- The C++ TRY block dynamically generates, compiles, and runs a tiny program for each
  expression \u2014 all without leaving C++.
- CATCH block does **not** reference `<expr_error` \u2014 it prints the literal `"try_block_failed"` string.
- Do **not** reference `result` or `insight` outside the TRY block \u2014 if TRY fails they were
  never written.
- `std::exit(1)` in the compile-failure branch is the idiomatic way to trigger CATCH from C++.
1 + 2 * 3

int x = @@@INVALID@@@

std::string(5, 'X')

[foreach] iter 1: expr=1 + 2 * 3

[try] executing try block

7

This is an integer literal value 7, which could be produced by arithmetic operations like addition (3 + 4), bitwise operations (15 & 7), or simply evaluating the literal constant 7 itself.

[OK] This is an integer literal value 7, which could be produced by arithmetic operations like addition (3 + 4), bitwise operations (15 & 7), or simply evaluating the literal constant 7 itself.

[try] try block succeeded

[foreach] iter 2: expr=int x = @@@INVALID@@@

[try] executing try block

/tmp/msh_p24_expr.cpp:4:25: error: stray \u2018@\u2019 in program

```
   4 |    auto val = (int x = @@@INVALID@@@);
     |                        ^
```

/tmp/msh_p24_expr.cpp:4:26: error: stray \u2018@\u2019 in program

```
   4 |    auto val = (int x = @@@INVALID@@@);
     |                         ^
```

/tmp/msh_p24_expr.cpp:4:27: error: stray \u2018@\u2019 in program

```
   4 |    auto val = (int x = @@@INVALID@@@);
     |                          ^
```

/tmp/msh_p24_expr.cpp:4:35: error: stray \u2018@\u2019 in program

```
   4 |    auto val = (int x = @@@INVALID@@@);
     |                                   ^
```

/tmp/msh_p24_expr.cpp:4:36: error: stray \u2018@\u2019 in program

```
   4 |    auto val = (int x = @@@INVALID@@@);
     |                                    ^
```

/tmp/msh_p24_expr.cpp:4:37: error: stray \u2018@\u2019 in program

4 | auto val = (int x = @@@INVALID@@@);

| ^

/tmp/msh_p24_expr.cpp: In function \u2018int main()\u2019:

/tmp/msh_p24_expr.cpp:4:17: error: expected primary-expression before \u2018int\u2019

4 | auto val = (int x = @@@INVALID@@@);

| ^~~

/tmp/msh_p24_expr.cpp:4:17: error: expected \u2018)\u2019 before \u2018int\u2019

4 | auto val = (int x = @@@INVALID@@@);

| ~^~~

|)

This is an integer value 7, which could be produced by operations such as arithmetic (like 3 + 4 or 14 / 2), bitwise manipulation (like 15 & 7), or simply evaluating the integer literal 7 directly.

[try] try block failed, executing catch block

[ERR] Expression failed: try_block_failed

[foreach] iter 3: expr=std::string(5, 'X')

[try] executing try block

XXXXX

This is a string value "XXXXX" consisting of five consecutive 'X' characters, which could be produced by string literal initialization, string concatenation operations, or string construction/manipulation functions like std::string(5, 'X').

[OK] This is a string value "XXXXX" consisting of five consecutive 'X' characters, which could be produced by string literal initialization, string concatenation operations, or string construction/manipulation functions like std::string(5, 'X').

[try] try block succeeded

/home/igor > === Batch complete. Errors were isolated, pipeline never stopped. ===

References:

Mshell Workflow Patterns – C++ Edition — Complete Reference Guide (P1–P24) — Art2Dec SoftLab (Non-profitable SoftLab), March 20th, 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/
