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

Pure Rust language Edition — Art2Dec SoftLab, March 22nd 2026

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

What is mshell?

mshell is a polyglot UNIX shell environment for AI and data workflows. It integrates multiple programming languages and LLM models into a single unified execution pipeline. Each code block is compiled and executed automatically. Variables flow between blocks via named files. LLM directives inject model responses into the pipeline as first-class variables.

Variable System

Reading a variable:

```
use std::{env, fs};
let val = fs::read_to_string(env::var("MSH_VAR_varname").unwrap()).unwrap();
```

Writing a variable (multiple >outvar block):

```
fs::write(env::var("MSH_VAR_varname").unwrap(), "value\n").unwrap();
// stdout is NOT captured when multiple >outvar are declared
```

Compiling and running LLM-generated Rust code:

```
use std::{fs, process::Command};
let src = "/tmp/gen.rs";
fs::write(src, &source).unwrap();
let c = Command::new("rustc").args([src, "-o",
"/tmp/gen_bin"]).output().unwrap();
if c.status.success() { Command::new("/tmp/gen_bin").status().unwrap(); }
```

Safe variable read (unwrap_or):

```
let val = env::var("MSH_VAR_name").ok()
    .and_then(|p| fs::read_to_string(&p).ok())
    .unwrap_or_else(|| "[not available]".to_string());
```

WHILE loop counter update:

```rust
let path = env::var("MSH_VAR_counter").unwrap();
let val: u64 = fs::read_to_string(&path).unwrap_or_else(|_| "0".into())
    .trim().parse().unwrap_or(0);
fs::write(&path, format!("{}", val + 1)).unwrap();
```

Async + Await barrier (use bash, not rust):

```bash
for p in "$(printenv MSH_VAR_ans1)" "$(printenv MSH_VAR_ans2)"; do
    i=0; while [ ! -s "$p" ] && [ $i -lt 180 ]; do sleep 1; i=$((i+1)); done
done
```

FOREACH list (no trailing newline):

```rust
fn main() {
    print!("item1\nitem2\nitem3");  // print! not println! — no trailing
newline
}
```

Critical Rules

1. **`bash await=`** — Always use `bash` blocks for async barriers, not `rust`. Rust blocks compile fresh each time and have a race condition.
2. **`bash <vars`** — After an await barrier, use `bash` to display async results (avoids compile race).
3. **Multiple `>outvar`** — Write directly to `env::var("MSH_VAR_*")` via `fs::write`. Stdout is NOT captured.
4. **WHILE counter** — Use `fs::read_to_string` / `fs::write` directly on `MSH_VAR_*` path.
5. **FOREACH list** — Use `print!` (no trailing newline). One item per line.
6. **TRY/CATCH** — Trigger CATCH with `std::process::exit(1)`. CATCH block prints literal `"try_block_failed"` — do NOT use `<errvar`.
7. **LLM prompts** — Add `You MUST respond` to prevent empty responses.
8. **Unicode in `println!`** — Use `\u{XXXX}` not `\uXXXX` (Rust requires braces).
9. **Safe reads** — Use `.unwrap_or_else(|| "[not available]".to_string())` in consumer blocks.
10. **Copy before compile** — Use `fs::copy(&code_path, "/tmp/stable.rs")` before `rustc` (MSH_VAR path may not have `.rs` extension).

Pattern Summary Table

#	Pattern	Nodes	Models	Key Technique
1	Linear Data Pipeline	—	—	Multi-stage Rust pipeline
2	LLM in the Middle	—	@1	Rust → LLM → Rust

#	Pattern	Nodes	Models	Key Technique
3	Fan-Out	—	@1	One var, multiple consumers
4	LLM Code Gen + Execute	—	@1	rustc compile LLM output
5	Two-LLM Review Chain	—	@1, @2	Generate → Review → Improve
6	Parallel 3-Model Query	—	@1–@3	Sequential LLM directives
7	Evaluator-Optimizer Loop	LOOP	@1, @2	ACCEPTED/REJECTED gate
8	Multi-Stage + Multi-Model	—	@1, @2	Rust stats + LLM analysis
9	Routing	—	@1	if=route:VALUE conditional
10	Full Pipeline	AWAIT	@1–@3	All patterns combined
11	MShell Native AI	—	@1, @2	ollama1/ollama2 inline
12	Async 3 Models + Synthesis	AWAIT	@1–@3	Parallel async + barrier
13	WHILE Counter	WHILE	@1	fs::write MSH_VAR_status
14	FOREACH List	FOREACH	@1	print! no trailing newline
15	TRY/CATCH	TRY/CATCH	—	exit(1) triggers CATCH
16	SPLIT + MERGE	SPLIT, MERGE	@1, @2	Async parallel + synthesis
17	CONFIG Pipeline	CONFIG	@1, @2	CONFIG docs, rust sets values
18	FOREACH + Async	FOREACH, AWAIT	@1, @2	Parallel per-item LLMs
19	WHILE Quality Gate	WHILE	@1, @2	fs::write status on threshold
20	Map-Reduce	SPLIT, MERGE, AWAIT	@1, @2	3 async map + reduce
21	TRY/CATCH + LOOP	TRY/CATCH, LOOP	@1	Self-correcting codegen
22	Multi-Variable Output	—	@1, @2	fs::write per MSH_VAR_*
23	CONFIG + WHILE + 3M	CONFIG, WHILE	@1–@3	Adaptive quality pipeline
24	FOREACH + TRY/CATCH	FOREACH	@1	Fault-tolerant batch compile

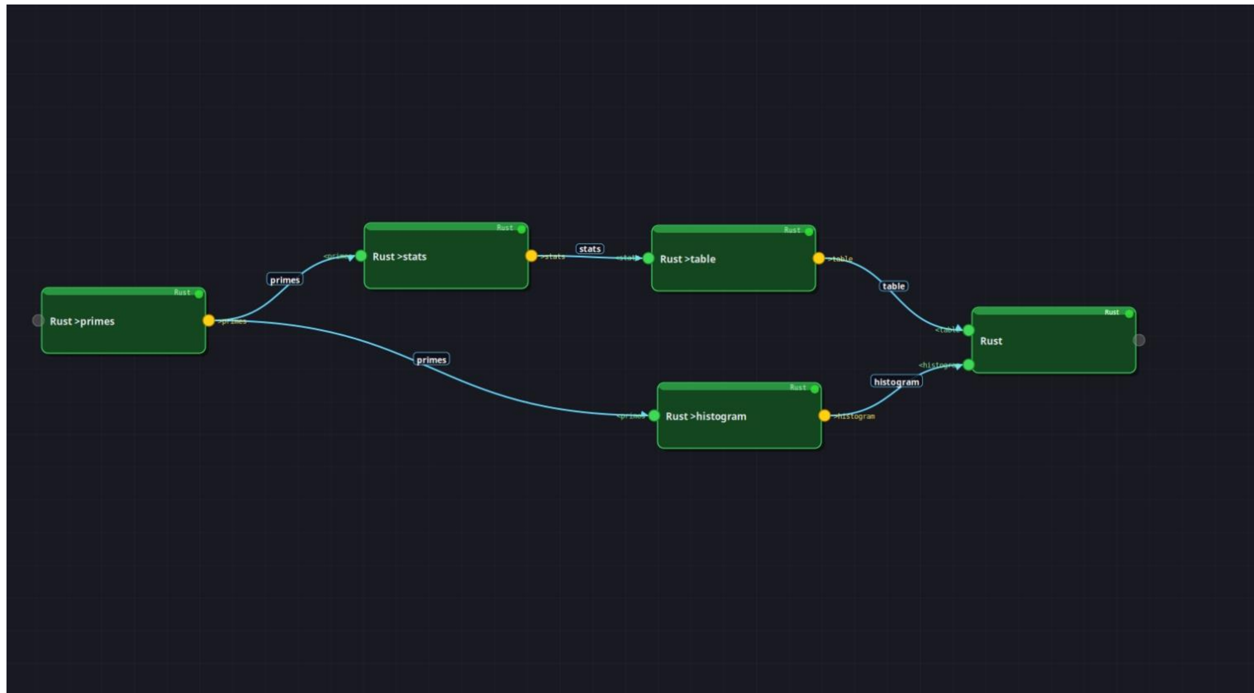
Part I — Patterns 1–12: Core Patterns

Pattern 1 — Linear Data Pipeline (Rust All Stages)

```
fn sieve(limit: usize) -> Vec<usize> {
    let mut is_prime = vec![true; limit + 1];
    is_prime[0] = false;
    if limit > 0 { is_prime[1] = false; }
    let mut i = 2;
    while i * i <= limit {
        if is_prime[i] {
            let mut j = i * i;
            while j <= limit { is_prime[j] = false; j += i; }
        }
        i += 1;
    }
    is_prime.iter().enumerate().filter(|(_, &p)| p).map(|(n, _)| n).collect()
}

fn main() {
    let first20: Vec<usize> = sieve(80).into_iter().take(20).collect();
    let out: Vec<String> = first20.iter().map(|n| n.to_string()).collect();
    println!("{}", out.join(","));
}

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_primes").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let nums: Vec<f64> = raw.trim().split(',').map(|s|
s.parse().unwrap()).collect();
    let n = nums.len() as f64;
    let sum: f64 = nums.iter().sum();
    let mean = sum / n;
    let variance = nums.iter().map(|x| (x - mean).powi(2)).sum::<f64>() / n;
    let mut sorted = nums.clone();
    sorted.sort_by(|a, b| a.partial_cmp(b).unwrap());
    let median = if sorted.len() % 2 == 0 {
        (sorted[sorted.len()/2 - 1] + sorted[sorted.len()/2]) / 2.0
    } else { sorted[sorted.len()/2] };
    println!("count={} sum={} mean={:.2} variance={:.2} median={} min={}
max={}",
        nums.len(), sum as usize, mean, variance,
        median as usize, sorted[0] as usize, sorted[sorted.len()-1] as
usize);
}

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_stats").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
```

```
println!("\u{250c}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{
2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}
\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{25
00}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u
{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2510}");
    println!("\u{2502}         Prime Numbers \u{2014} Statistics
\u{2502}");

println!("\u{251c}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{
2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{252c}\u{2500}\u{2500}
\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{25
00}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u
{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2524}");
    for part in raw.trim().split_whitespace() {
        if let Some((k, v)) = part.split_once('=') {
            println!("\u{2502} {:12} \u{2502} {:>24} \u{2502}", k, v);
        }
    }

println!("\u{2514}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{
2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2534}\u{2500}\u{2500}
\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{25
00}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u
{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2518}");
}

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_primes").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let nums: Vec<usize> = raw.trim().split(',').map(|s|
s.parse().unwrap()).collect();
    let mut freq = [0usize; 10];
    for &n in &nums { freq[n % 10] += 1; }
    println!("Last-digit frequency histogram:");
    for (digit, &count) in freq.iter().enumerate() {
        if count > 0 { println!("  {} | {}", digit, "\u{2588}".repeat(count *
4)); }
    }
}

use std::{env, fs};
fn main() {
    for var in &["table", "histogram"] {
        let path = env::var(&format!("MSH_VAR_{}", var)).unwrap();
        println!("{}", fs::read_to_string(&path).unwrap().trim());
        println!();
    }
```

```
    println!("=== Pipeline complete ===");
}
```

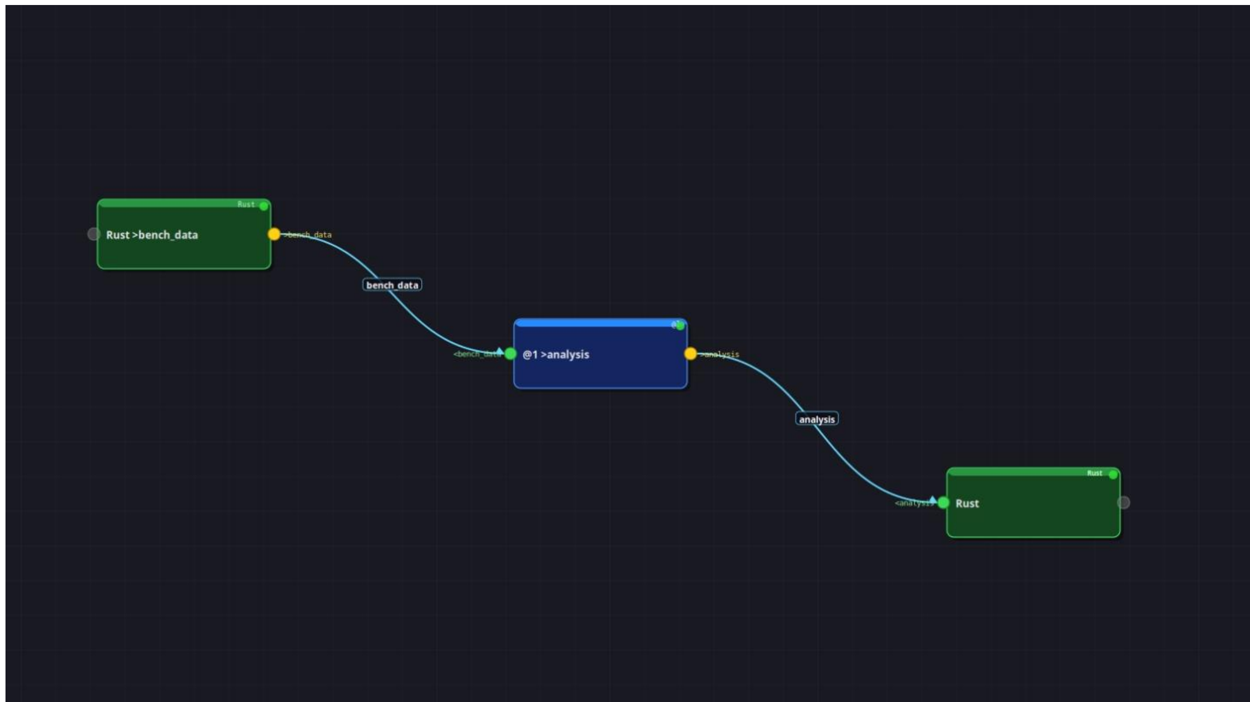


```
Terminal
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71
count=20 sum=639 mean=31.95 variance=479.55 median=30 min=2 max=71
┌──────────────────────────────────────────┐
│        Prime Numbers — Statistics        │
├──────────────┬───────────────────────────┤
│ count        │                        20 │
│ sum          │                       639 │
│ mean         │                     31.95 │
│ variance     │                    479.55 │
│ median       │                        30 │
│ min          │                         2 │
│ max          │                        71 │
└──────────────┴───────────────────────────┘
Last-digit frequency histogram:
  1 |
  2 |
  3 |
  5 |
  7 |
  9 |
/home/igor >
        Prime Numbers — Statistics
  count                                  20
  sum                                   639
  mean                                31.95
  variance                           479.55
  median                                 30
  min                                     2
  max                                    71

Last-digit frequency histogram:
  1 |
  2 |
  3 |
  5 |
  7 |
  9 |

=== Pipeline complete ===

/home/igor >
```

Pattern 2 — LLM in the Middle (Rust + LLM + Rust)

```
fn main() {
    let metrics = [
        ("integer_ops_per_sec",   "4_820_000_000"),
        ("float_ops_per_sec",     "2_310_000_000"),
        ("memory_bandwidth_gbs", "48.7"),
        ("cache_miss_rate_pct",   "3.2"),
        ("p50_latency_us",        "12"),
```

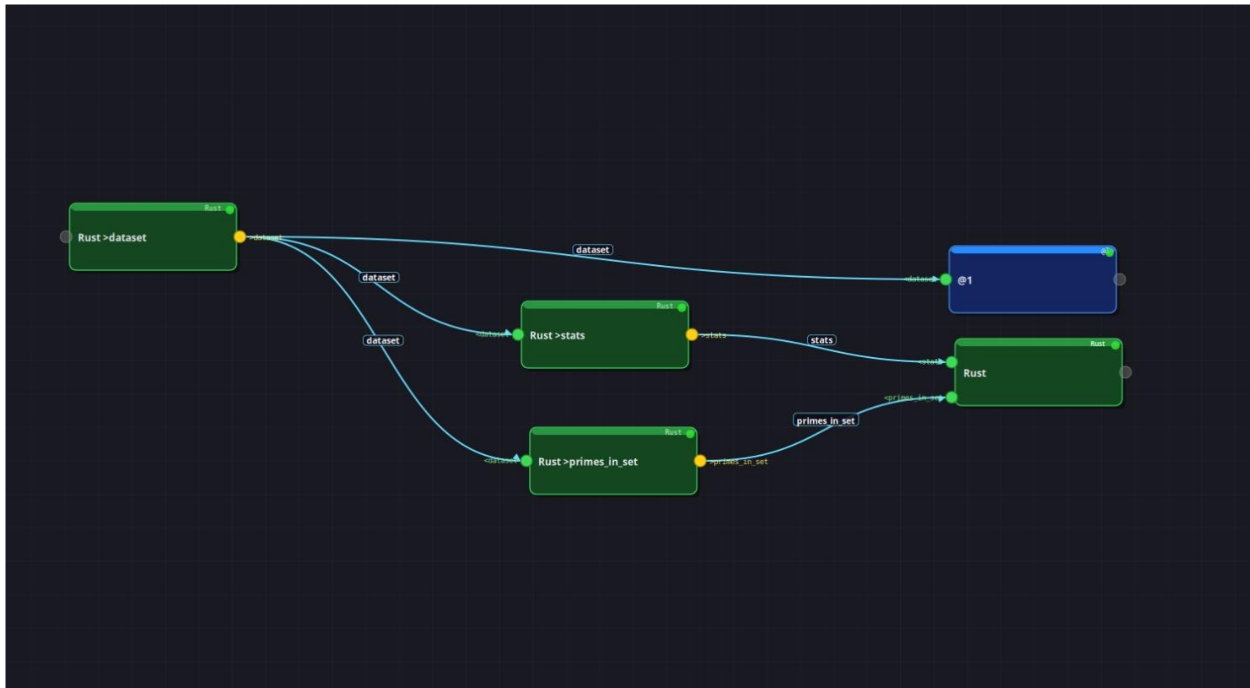
```
        ("p95_latency_us",       "87"),
        ("p99_latency_us",       "214"),
        ("thread_count",         "16"),
    ];
    println!("=== Benchmark Report ===");
    for (k, v) in &metrics { println!("{}: {}", k, v); }
}

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_analysis").unwrap();
    let text = fs::read_to_string(&path).unwrap();
    let text = text.trim();
    let words = text.split_whitespace().count();
    let sentences = text.chars().filter(|&c| matches!(c, '.' | '!' |
'?')).count();
    let paras = text.split("\n\n").filter(|s| !s.trim().is_empty()).count();

println!("\u{2554}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{
2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}
\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{25
50}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u
{2550}\u{2550}\u{2557}");
    println!("\u{2551}      LLM Performance Analysis        \u{2551}");

println!("\u{2560}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{
2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}
\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{25
50}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u
{2550}\u{2550}\u{2563}");
    println!("\u{2551} Words    : {:>28} \u{2551}", words);
    println!("\u{2551} Sentences: {:>27} \u{2551}", sentences);
    println!("\u{2551} Paragraphs: {:>26} \u{2551}", paras);

println!("\u{255a}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{
2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}
\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{25
50}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u
{2550}\u{2550}\u{255d}");
    println!("\n{}", text);
}
```



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

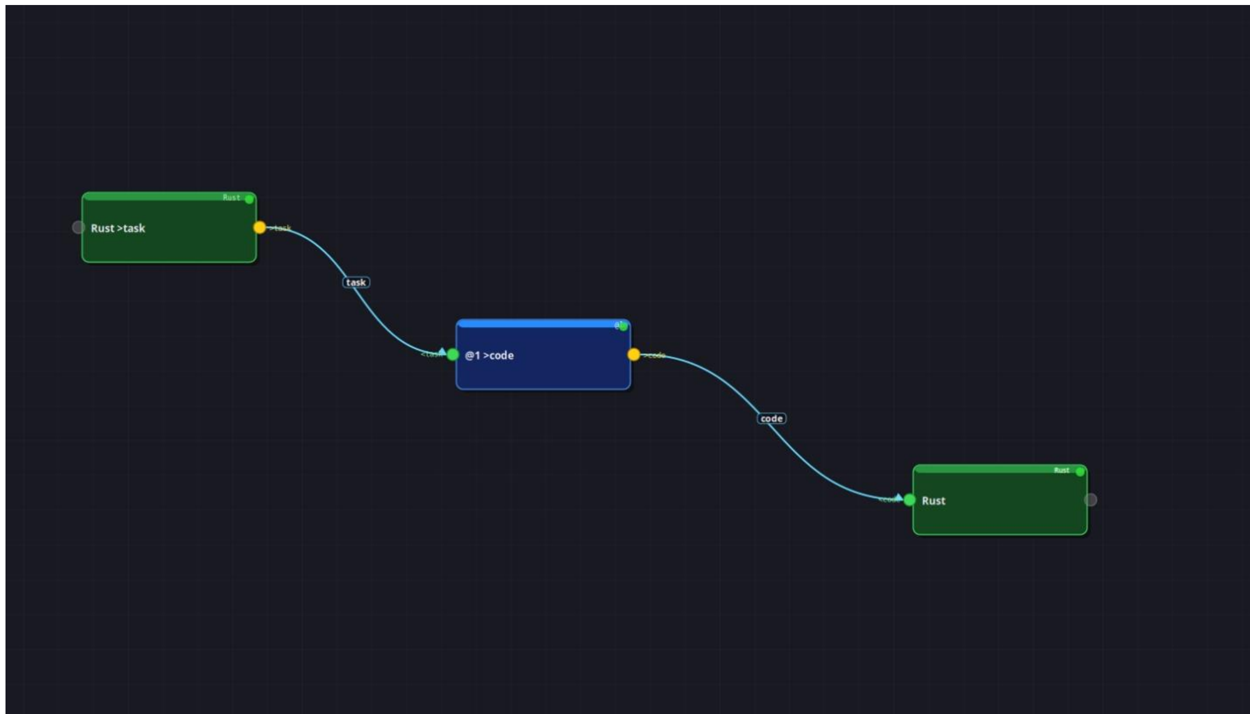
```
fn main() {
    let mut state: u64 = 0xDEAD_BEEF_1234_5678;
    let mut nums = Vec::new();
    for _ in 0..15 {
        state =
state.wrapping_mul(6364136223846793005).wrapping_add(1442695040888963407);
        nums.push((state >> 33) % 200 + 2);
    }
    println!("{}", nums.iter().map(|n|
n.to_string()).collect::<Vec<_>>().join(" "));
}

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_dataset").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let nums: Vec<f64> = raw.trim().split_whitespace().map(|s|
s.parse().unwrap()).collect();
    let n = nums.len() as f64;
    let mean = nums.iter().sum::<f64>() / n;
    let std_dev = (nums.iter().map(|x| (x - mean).powi(2)).sum::<f64>() /
n).sqrt();
    let mut sorted = nums.clone();
    sorted.sort_by(|a, b| a.partial_cmp(b).unwrap());
    println!("n={} mean={:.1} std_dev={:.1} min={} max={} median={}",
        nums.len(), mean, std_dev,
        sorted[0] as usize, sorted[sorted.len()-1] as usize,
```

```
        sorted[sorted.len()/2] as usize);
}

use std::{env, fs};
fn is_prime(n: u64) -> bool {
    if n < 2 { return false; }
    if n == 2 { return true; }
    if n % 2 == 0 { return false; }
    let mut i = 3u64;
    while i * i <= n { if n % i == 0 { return false; } i += 2; }
    true
}
fn main() {
    let path = env::var("MSH_VAR_dataset").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let primes: Vec<u64> = raw.trim().split_whitespace()
        .map(|s| s.parse().unwrap()).filter(|&n| is_prime(n)).collect();
    println!("Primes found ({}): {}", primes.len(),
        primes.iter().map(|n| n.to_string()).collect::<Vec<_>>().join(", "));
}

use std::{env, fs};
fn main() {
    for (label, var) in &[("Statistics", "stats"), ("Primes in set",
"primes_in_set")] {
        let path = env::var(&format!("MSH_VAR_{}", var)).unwrap();
        println!("--- {} ---\n{}\n", label,
fs::read_to_string(&path).unwrap().trim());
    }
}
```



Pattern 4 — LLM Code Generation → Execute via Variable

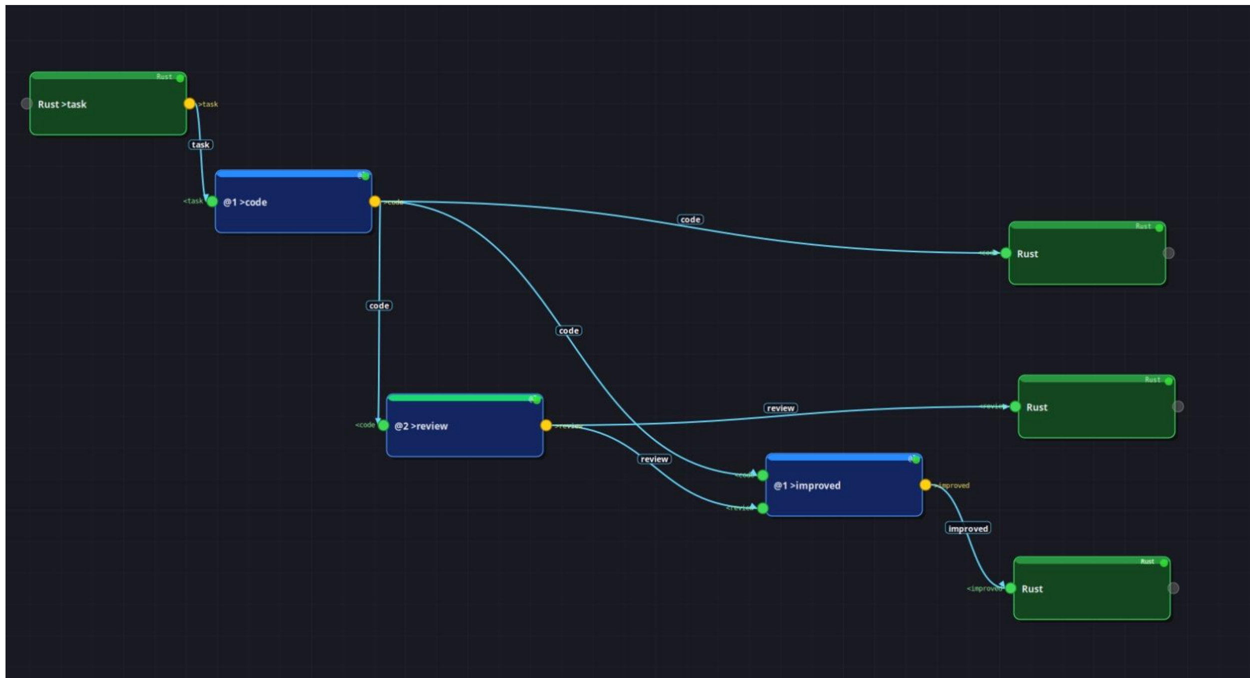
```
fn main() {
    println!("Write a Rust program that generates Fibonacci numbers up to 1000, \
        filters only those divisible by 3, and prints each with its index and running sum.");
}

use std::{env, fs, process::Command};
fn main() {
    let code_path = env::var("MSH_VAR_code").unwrap();
    let source = fs::read_to_string(&code_path).unwrap();
    println!("=== Generated Rust Code ===\n{}\n", source.trim());
    let src = "/tmp/mshell_generated.rs";
    let bin = "/tmp/mshell_generated_bin";
    fs::write(src, &source).unwrap();
    let compile = Command::new("rustc").args([src, "-o", bin]).output().unwrap();
    if !compile.status.success() {
        eprintln!("{}", String::from_utf8_lossy(&compile.stderr));
        return;
    }
    println!("=== Output ===");
    print!("{}", String::from_utf8_lossy(&Command::new(bin).output().unwrap().stdout));
}
```



Pattern 5 — Two-LLM Review Chain

```
fn main() {
    println!("Write a Rust function that counts word frequencies in a string \
        using HashMap, returns top-3 most frequent words sorted by frequency desc \
        then alphabetically. Include fn main() with a demo.");
}

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_code").unwrap();
    println!("=== Model @1 Generated ===\n{}\n",
        fs::read_to_string(&path).unwrap().trim());
}

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_review").unwrap();
    println!("=== Model @2 Review ===\n{}\n",
        fs::read_to_string(&path).unwrap().trim());
}

use std::{env, fs, process::Command};
fn main() {
    let code_path = env::var("MSH_VAR_improved").unwrap();
    let source = fs::read_to_string(&code_path).unwrap();
    println!("=== Final Improved Code ===\n{}\n", source.trim());
    let src = "/tmp/mshell_improved.rs";
```

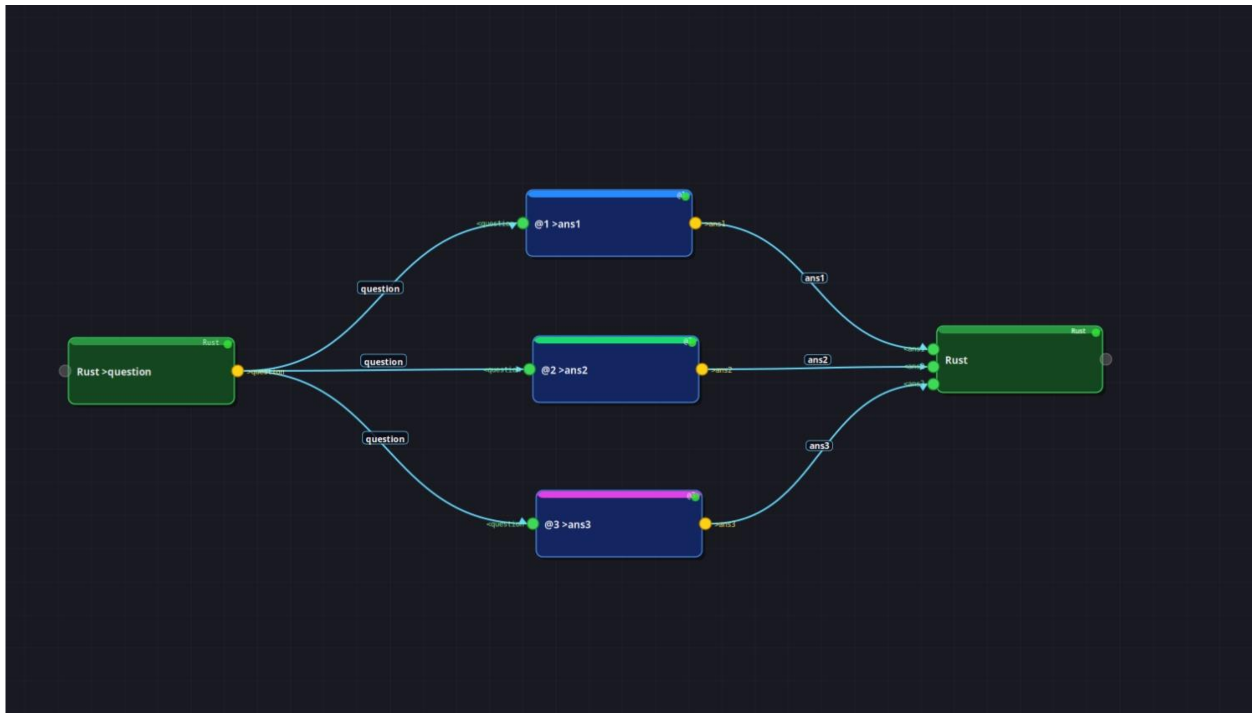
```
    let bin = "/tmp/mshell_improved_bin";
    fs::write(src, &source).unwrap();
    let c = Command::new("rustc").args([src, "-o", bin]).output().unwrap();
    if !c.status.success() { eprintln!("{}",
String::from_utf8_lossy(&c.stderr)); return; }
    println!("=== Executing ===");
    print!("{}",
String::from_utf8_lossy(&Command::new(bin).output().unwrap().stdout));
}
```



Pattern 6 — Parallel 3-Model Query

```
fn main() {
    println!("In Rust, for a high-throughput network service, what is the \
        most important architectural decision regarding async runtime and \
        concurrency model? Answer in exactly two sentences.");
}

use std::{env, fs};
fn main() {
    for (label, var) in &[
        ("@1 Production Focus", "ans1"),
        ("@2 Performance Focus", "ans2"),
        ("@3 Maintainability Focus", "ans3"),
    ] {
        let path = env::var(&format!("MSH_VAR_{}", var)).unwrap();
        let content = fs::read_to_string(&path).unwrap();
        println!("\u{250c}\u{2500} {} {}", label,
"\u{2500}".repeat(44usize.saturating_sub(label.len())));
        println!("\u{2502} {}", content.trim());
```

```
        println!("\u{2514}{}\n", "\u{2500}".repeat(50));
    }
}
```



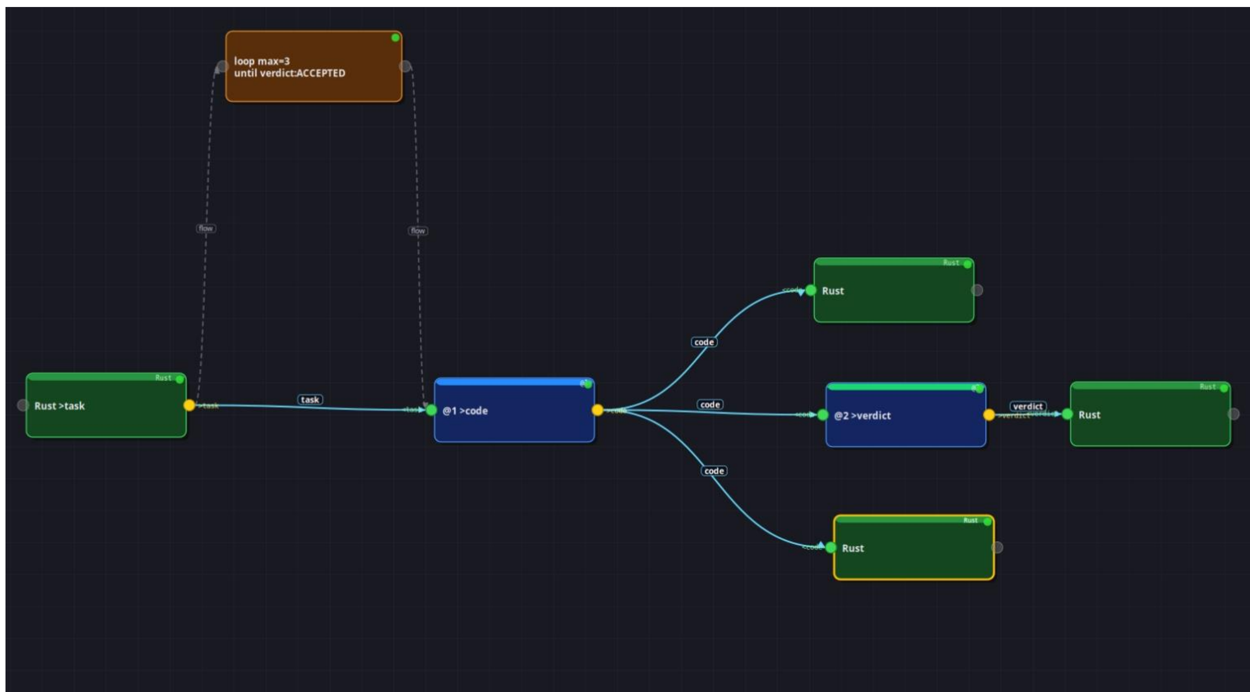
Pattern 7 — Evaluator-Optimizer Loop

```
fn main() {
    println!("Write generic Rust fn binary_search<T: Ord>(slice: &[T],
target: &T) \
        -> Option<usize>. Include fn main() with tests for: found, not-found,
\
        empty slice, single element, duplicates. No
std::slice::binary_search.");
}
```

```
use std::{env, fs};
fn main() {
    println!("=== Generated ===\n{}\n",

fs::read_to_string(env::var("MSH_VAR_code").unwrap()).unwrap().trim());
}
```

```
use std::{env, fs};
fn main() {
    println!("=== Verdict: {} ===\n",

fs::read_to_string(env::var("MSH_VAR_verdict").unwrap()).unwrap().trim());
}
```

```
use std::{env, fs, process::Command};
fn main() {
    let source =
fs::read_to_string(env::var("MSH_VAR_code").unwrap()).unwrap();
    println!("=== Final Accepted Code ===\n{}\n", source.trim());
    let src = "/tmp/mshell_p7_final.rs";
    let bin = "/tmp/mshell_p7_final_bin";
    fs::write(src, &source).unwrap();
    let c = Command::new("rustc").args([src, "-o", bin]).output().unwrap();
    if !c.status.success() {
        eprintln!("Compile error:\n{}", String::from_utf8_lossy(&c.stderr));
        return;
    }
    println!("=== Executing ===");
    print!("{}",
String::from_utf8_lossy(&Command::new(bin).output().unwrap().stdout));
}
```

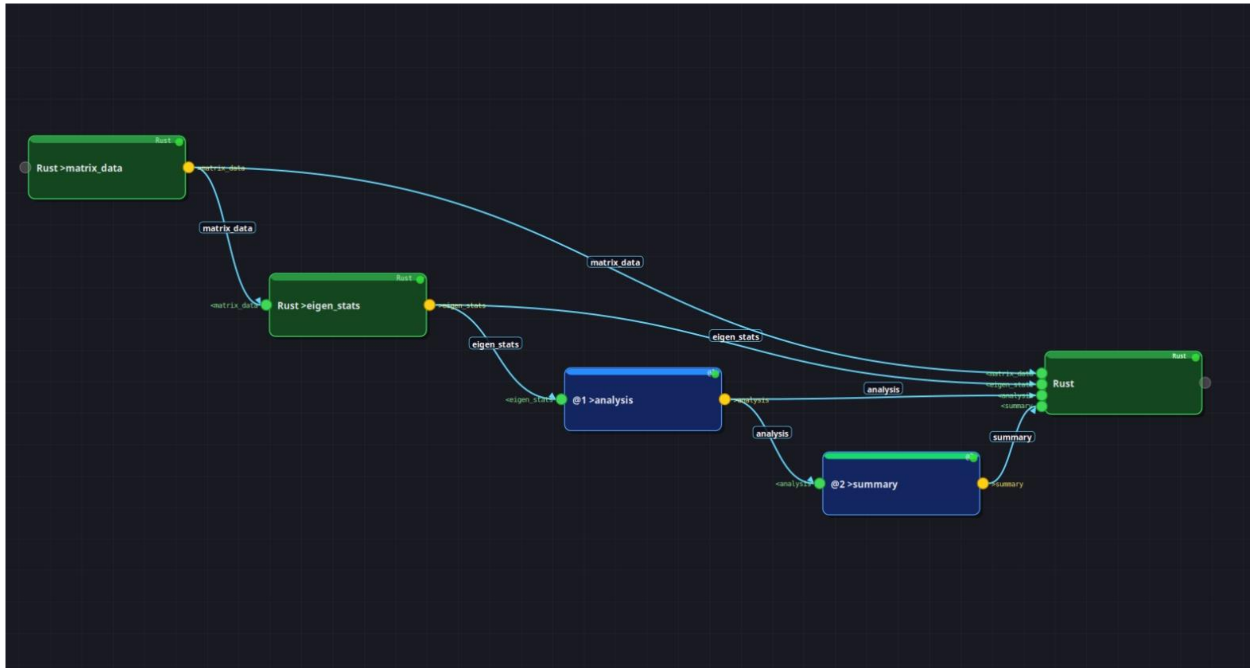


Pattern 8 — Multi-Stage Rust + Multi-Model

```
fn det3(a: [[f64;3];3]) -> f64 {
    a[0][0]*(a[1][1]*a[2][2]-a[1][2]*a[2][1])
   -a[0][1]*(a[1][0]*a[2][2]-a[1][2]*a[2][0])
   +a[0][2]*(a[1][0]*a[2][1]-a[1][1]*a[2][0])
}
fn det4(a: [[f64;4];4]) -> f64 {
    (0..4).map(|j| {
        let sign = if j%2==0 { 1.0 } else { -1.0 };
        let mut minor = [[0.0f64;3];3];
        for r in 1..4 { let mut ci=0; for c in 0..4 { if c==j { continue; }
```

```
            minor[r-1][ci]=a[r][c]; ci+=1; } }
        sign * a[0][j] * det3(minor)
    }).sum()
}
fn main() {
    let m: [[f64;4];4] =
[[4.,3.,2.,1.],[3.,6.,1.,2.],[2.,1.,8.,3.],[1.,2.,3.,9.]];
    println!("determinant={:.4}", det4(m));
    println!("trace={:.1}", m[0][0]+m[1][1]+m[2][2]+m[3][3]);
    println!("values={}", (0..4).flat_map(|r| (0..4).map(move |c|
m[r][c].to_string()))
        .collect::<Vec<_>>().join(","));
}

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_matrix_data").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let mut vals = vec![];
    for line in raw.lines() {
        if let Some(v) = line.strip_prefix("values=") {
            vals = v.split(',').map(|s|
s.parse::<f64>().unwrap()).collect::<Vec<_>>();
        }
    }
    let m: Vec<Vec<f64>> = (0..4).map(|r|
vals[r*4..r*4+4].to_vec()).collect();
    let mut v = vec![1.0f64; 4];
    let mut eigenvalue = 0.0f64;
    for _ in 0..50 {
        let mut mv = vec![0.0f64; 4];
        for i in 0..4 { for j in 0..4 { mv[i] += m[i][j] * v[j]; } }
        eigenvalue = mv.iter().cloned().fold(f64::NEG_INFINITY, f64::max);
        let norm = mv.iter().map(|x| x*x).sum::<f64>().sqrt();
        v = mv.iter().map(|x| x/norm).collect();
    }
    println!("dominant_eigenvalue={:.4}", eigenvalue);
    println!("frobenius_norm={:.4}", vals.iter().map(|x|
x*x).sum::<f64>().sqrt());
}

use std::{env, fs};
fn slurp(var: &str) -> String {
    env::var(var).ok()
        .and_then(|p| fs::read_to_string(&p).ok())
        .unwrap_or_else(|| "[not available]".to_string())
}
fn main() {
    for (label, var) in &[
        ("Matrix",   "MSH_VAR_matrix_data"),
```

```
        ("Eigen",    "MSH_VAR_eigen_stats"),
        ("Analysis", "MSH_VAR_analysis"),
        ("Summary",  "MSH_VAR_summary"),
    ] {
        println!("[{}]\n{}\n", label, slurp(var).trim());
    }
}
```

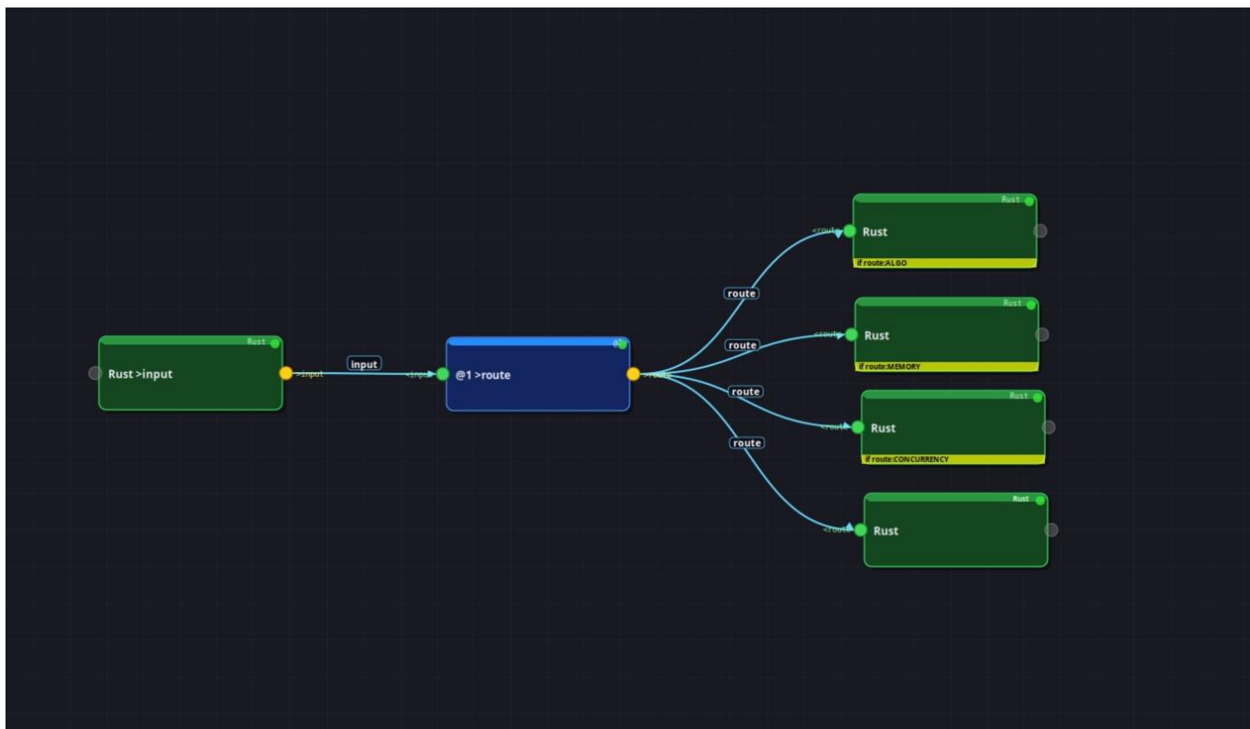


Pattern 9 — Routing: LLM Classifies → Conditional Rust Branch

```
fn main() {
    println!("What is the most efficient way to sort a large Vec<u64> \
        in Rust when data has many repeated values?");
}

use std::time::Instant;
fn main() {
    let mut data: Vec<u64> = (0..100_000).map(|i| i % 1000).collect();
    let t0 = Instant::now();
    data.sort_unstable();
    println!("=== ALGO: Sort Benchmark ===");
    println!("Sorted 100k elements in {:?}", t0.elapsed());
    println!("pdqsort optimal for low-cardinality data");
}

fn main() {
    println!("=== MEMORY: Allocation Decision Tree ===");
    println!("Box<T>  \u{2192} single owner, zero overhead");
    println!("Rc<T>   \u{2192} shared, single-thread");
    println!("Arc<T>  \u{2192} shared, multi-thread safe");
}
```

```rust
use std::{time::Instant, thread};
fn main() {
    let t0 = Instant::now();
    let h: Vec<_> = (0..100).map(|i| thread::spawn(move || i*2)).collect();
    for h in h { h.join().unwrap(); }
    println!("=== CONCURRENCY: Thread Benchmark ===");
    println!("100 OS threads spawn+join: {:?}", t0.elapsed());
}
```

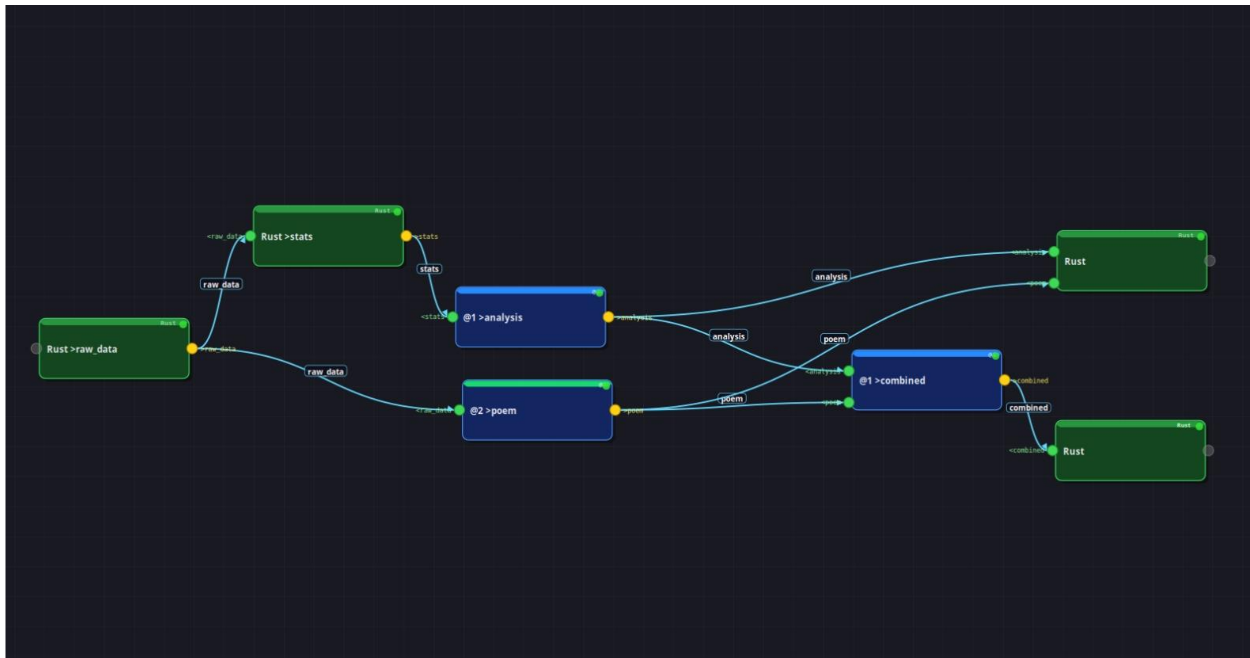
```rust
use std::{env, fs};
fn main() {
    let route = env::var("MSH_VAR_route").ok()
        .and_then(|p| fs::read_to_string(&p).ok())
        .unwrap_or_else(|| "[not classified]".to_string());
    println!("Classified as: {}", route.trim());
}
```



Pattern 10 — Full Pipeline: All Patterns Combined

```rust
fn is_prime(n: u64) -> bool {
    if n < 2 { return false; }
    if n == 2 || n == 3 { return true; }
    if n%2==0 || n%3==0 { return false; }
    let mut i = 5u64;
    while i*i <= n { if n%i==0 || n%(i+2)==0 { return false; } i+=6; }
    true
}
fn main() {
    let mersennes: Vec<u64> = (2u32..=19)
```

```
        .filter(|&p| is_prime(p as u64))
        .map(|p| (1u64 << p) - 1)
        .filter(|&m| is_prime(m))
        .collect();
    println!("{}", mersennes.iter().map(|n|
n.to_string()).collect::<Vec<_>>().join(","));
}

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_raw_data").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let nums: Vec<u64> = raw.trim().split(',').map(|s|
s.trim().parse().unwrap()).collect();
    let sum: u64 = nums.iter().sum();
    println!("count={} sum={} min={} max={}", nums.len(), sum,
        nums.iter().min().unwrap(), nums.iter().max().unwrap());
    let forms: Vec<String> = nums.iter().map(|&n| {
        let p = (n+1).next_power_of_two().trailing_zeros();
        format!("2^{}-1", p)
    }).collect();
    println!("forms={}", forms.join(", "));
}

use std::{env, fs};
fn main() {
    for (l, v) in &[("Analysis","analysis"),("Poem","poem")] {
        println!("=== {} ===\n{}\n", l,
            fs::read_to_string(env::var(&format!("MSH_VAR_{}",
v)).unwrap()).unwrap().trim());
    }
}

use std::{env, fs};
fn main() {
    let text =
fs::read_to_string(env::var("MSH_VAR_combined").unwrap()).unwrap();
    let text = text.trim();
    let w = text.len().max(50) + 4;
    println!("\u{2554}{}\u{2557}", "\u{2550}".repeat(w-2));
    println!("\u{2551}{:^width$}\u{2551}", " Mersenne Primes \u{2014} Final
Synthesis ", width=w-2);
    println!("\u{2560}{}\u{2563}", "\u{2550}".repeat(w-2));
    println!("\u{2551}  {:<width$}  \u{2551}", text, width=w-6);
    println!("\u{255a}{}\u{255d}", "\u{2550}".repeat(w-2));
}
```



Pattern 11 — MShell Node with Multiple Models

```
fn main() { println!("Rust ownership and borrowing system"); }

fn main() { println!("precise and example-driven for systems programmers"); }

ollama1 "Explain '$topic' in a '$style' way in two sentences"

ollama2 "Extract exactly 4 keywords from: $explanation. Reply with only the 4
keywords comma-separated."

use std::{env, fs};
fn main() {
    let exp =
fs::read_to_string(env::var("MSH_VAR_explanation").unwrap()).unwrap();
    let kw  =
fs::read_to_string(env::var("MSH_VAR_keywords").unwrap()).unwrap();
    println!("=== Explanation ===\n{}\n", exp.trim());
    println!("=== Keywords ===\n{}", kw.trim());
}
```

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

```
fn main() {
    println!("What is the difference between dyn Trait and <T: Trait> in
Rust, \
        and when should you use each? Answer in one sentence.");
}

for p in "$(printenv MSH_VAR_ans1)" "$(printenv MSH_VAR_ans2)" "$(printenv
MSH_VAR_ans3)"; do
    i=0
    while [ ! -s "$p" ] && [ $i -lt 180 ]; do sleep 1; i=$((i+1)); done
    if [ ! -s "$p" ]; then echo "[no response]" > "$p"; fi
done

pf=$(printenv MSH_VAR_final)
i=0
while [ ! -s "$pf" ] && [ $i -lt 180 ]; do sleep 1; i=$((i+1)); done
if [ ! -s "$pf" ]; then echo "[synthesis not available]" > "$pf"; fi
echo "\u250c\u2500 Beginner Intuition (@1)"; cat $(printenv MSH_VAR_ans1);
echo
"\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\
u2500\u2500"
echo "\u250c\u2500 Memory Layout (@2)";      cat $(printenv MSH_VAR_ans2);
echo
"\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
```

```
2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\
u2500\u2500"
echo "\u250c\u2500 Type Theory (@3)";        cat $(printenv MSH_VAR_ans3);
echo
"\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\
u2500\u2500"
echo
"\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550"
cat $(printenv MSH_VAR_final)
echo
"\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550"
```

Part II — Patterns 13–24: Advanced Node Types

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

Pattern 13 — WHILE Loop: Iterative Counter with LLM Commentary

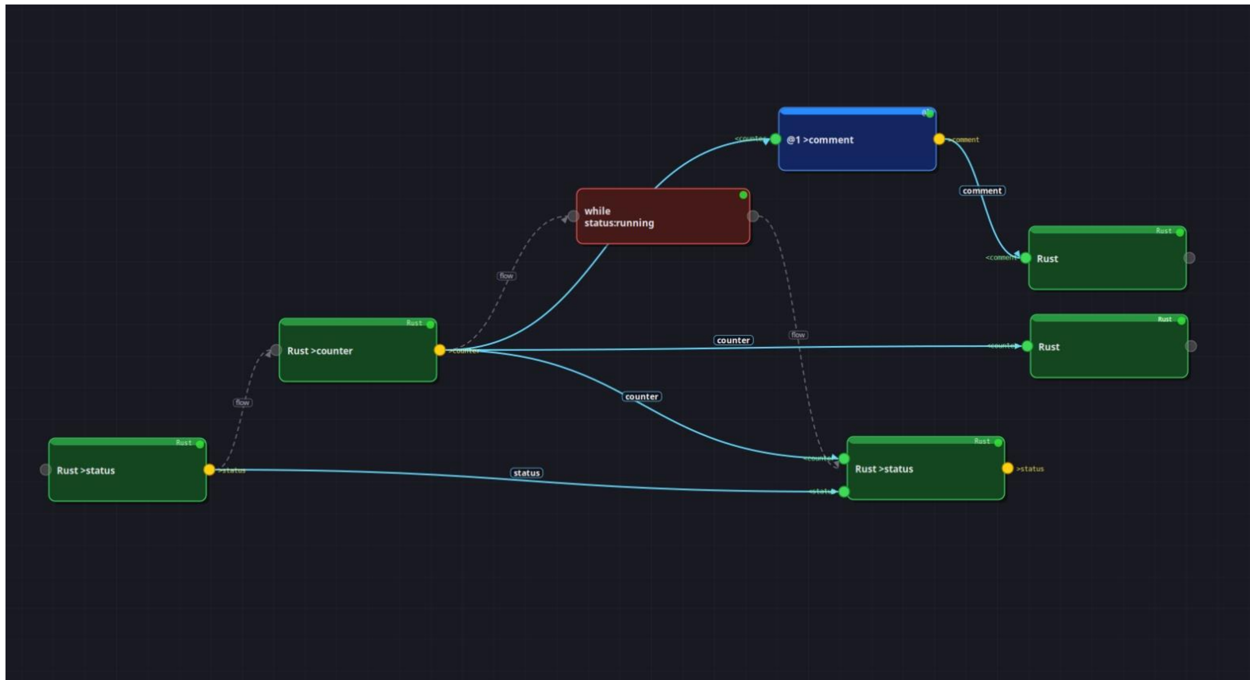
```
fn main() { println!("running"); }
```

```
fn main() { println!("0"); }
```

```
use std::{env, fs};
fn main() {
    let counter_path = env::var("MSH_VAR_counter").unwrap();
    let status_path  = env::var("MSH_VAR_status").unwrap();
    let raw = fs::read_to_string(&counter_path).unwrap_or_else(|_|
"0".into());
    let val: u64 = raw.trim().parse().unwrap_or(0);
    let new_val = val + 1;
    fs::write(&counter_path, format!("{}", new_val)).unwrap();
    fs::write(&status_path, if new_val >= 4 { "done" } else { "running"
}).unwrap();
    print!("{}", new_val);
}
```

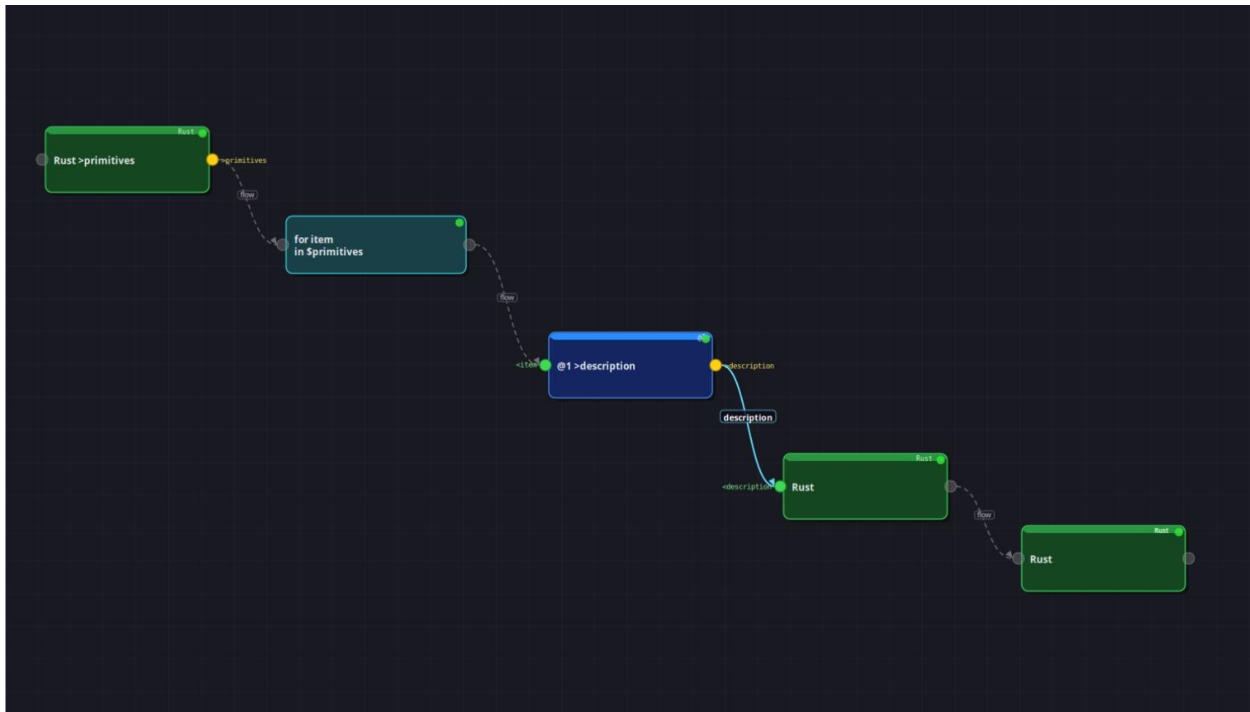
```
use std::{env, fs};
fn main() {
    let counter =
fs::read_to_string(env::var("MSH_VAR_counter").unwrap()).unwrap();
    let comment =
fs::read_to_string(env::var("MSH_VAR_comment").unwrap()).unwrap();
    println!("[iter {}] {}", counter.trim(), comment.trim());
}
```

```
use std::{env, fs};
fn main() {
    println!("=== WHILE complete. Final counter = {} ===",

fs::read_to_string(env::var("MSH_VAR_counter").unwrap()).unwrap().trim());
}
```



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

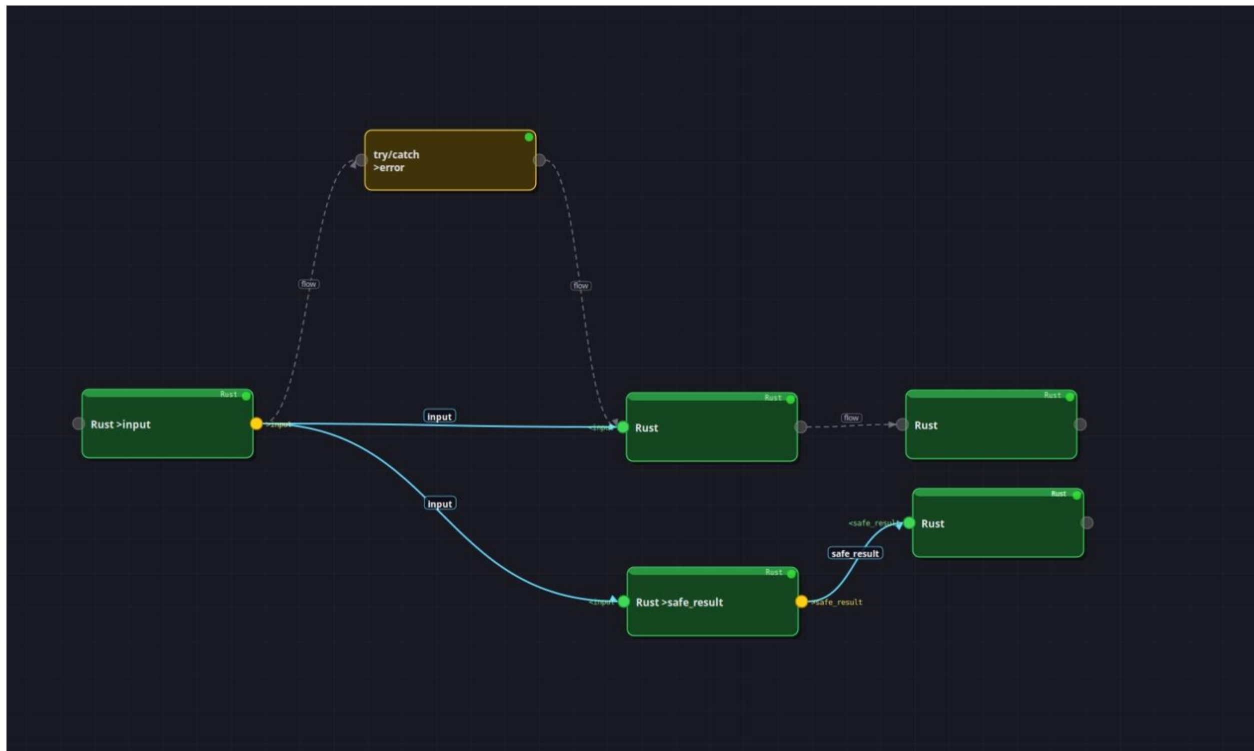
```
fn main() {
    print!("Mutex\nRwLock\nAtomicUsize\nMpscChannel\nBarrier");
}

use std::{env, fs};
fn main() {
    let item =
fs::read_to_string(env::var("MSH_VAR_item").unwrap()).unwrap();
    let desc =
fs::read_to_string(env::var("MSH_VAR_description").unwrap()).unwrap();
    let name = item.trim();
    println!("\u{250c}\u{2500} {} {}", name,
"\u{2500}".repeat(50usize.saturating_sub(name.len()+3)));
    println!("\u{2502} {}", desc.trim());
    println!("\u{2514}{}\n", "\u{2500}".repeat(52));
}

fn main() { println!("=== All 5 Rust concurrency primitives described ===");
}
```



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

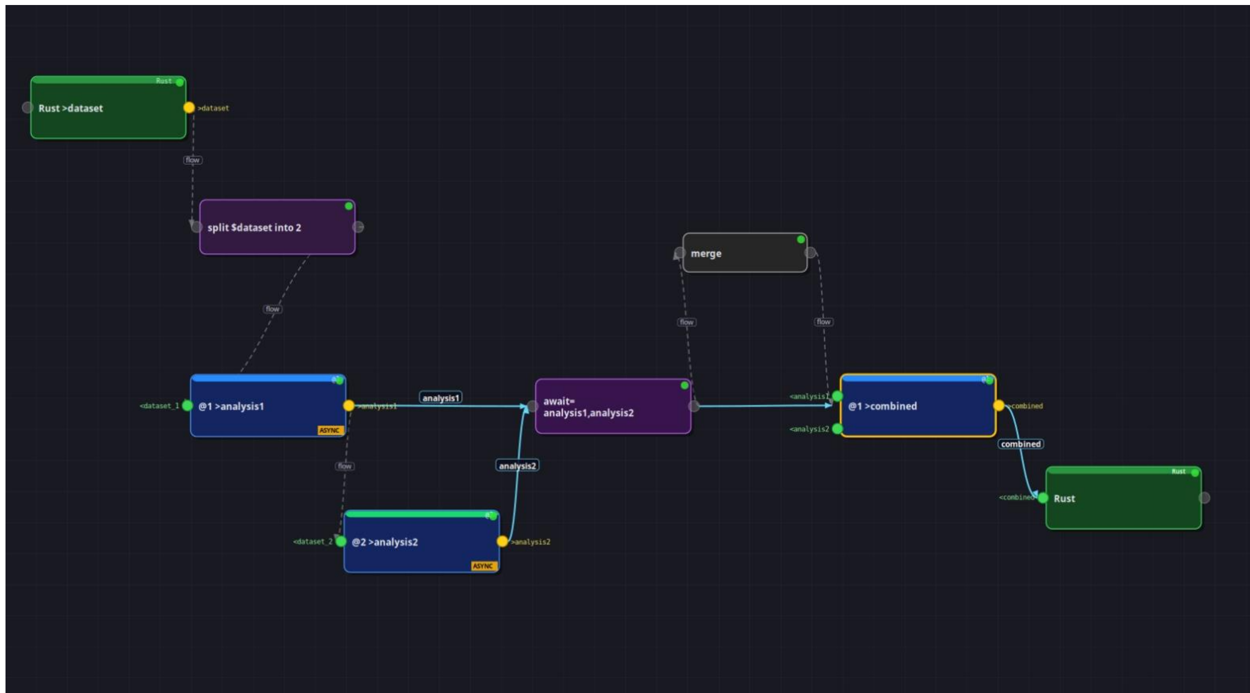
```
fn main() {
    println!(r#"{{name: "Ferris", lang: "Rust", version: 1.75"#);
}

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_input").unwrap();
    let s = fs::read_to_string(&path).unwrap();
    let s = s.trim();
    if !s.starts_with('{') || !s.ends_with('}') {
        eprintln!("JSON error: not a valid object");
        std::process::exit(1);
    }
    for part in s.trim_matches(|c| c == '{' || c == '}').split(',') {
        let kv: Vec<&str> = part.splitn(2, ':').collect();
        if kv.len() != 2 { eprintln!("JSON error: bad kv");
std::process::exit(1); }
        let key = kv[0].trim();
        if !key.starts_with('"') { eprintln!("JSON error: unquoted key: {}",
key); std::process::exit(1); }
    }
    println!("Parsed OK");
}

fn main() {
    println!("=== Caught: try_block_failed ===");
```

```
    println!("JSON parse failed. Activating safe fallback.");
}

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_input").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let cleaned = raw.trim().trim_matches(|c| c=='{' || c=='}').replace('"',
"");
    let fields: Vec<String> = cleaned.split(',').filter_map(|p| {
        let kv: Vec<&str> = p.splitn(2, ':').collect();
        if kv.len()==2 { Some(format!("{}={}", kv[0].trim(), kv[1].trim())) }
else { None }
    }).collect();
    println!("Safe fallback: {} fields: {}", fields.len(), fields.join(" |
"));
}

use std::{env, fs};
fn main() {
    println!("=== Safe Result ===\n{}",

fs::read_to_string(env::var("MSH_VAR_safe_result").unwrap()).unwrap().trim())
;
}
```



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

```
fn main() {
    print!("12,15,11,18,14,13,16,10,19,12\n45,38,52,41,47,50,43,39,55,44");
}

for p in "$(printenv MSH_VAR_analysis1)" "$(printenv MSH_VAR_analysis2)"; do
    i=0; while [ ! -s "$p" ] && [ $i -lt 180 ]; do sleep 1; i=$((i+1)); done
    if [ ! -s "$p" ]; then echo "[no response]" > "$p"; fi
done

use std::{env, fs};
fn stats(data: &str) -> (f64, f64) {
    let nums: Vec<f64> = data.trim().split(',').map(|s|
s.trim().parse().unwrap_or(0.0)).collect();
    let mean = nums.iter().sum::<f64>() / nums.len() as f64;
    let std_dev = (nums.iter().map(|x| (x-mean).powi(2)).sum::<f64>() /
nums.len() as f64).sqrt();
    (mean, std_dev)
}
fn main() {
    for (label, var) in &[("Suite A (Parser)","dataset_1"),("Suite B
(Serializer)","dataset_2")] {
        let raw = fs::read_to_string(env::var(&format!("MSH_VAR_{}",
var)).unwrap()).unwrap();
        let (mean, sd) = stats(&raw);
        println!("{}: mean={:.1}ns  \u{03c3}={:.1}ns", label, mean, sd);
    }
    let combined =
fs::read_to_string(env::var("MSH_VAR_combined").unwrap()).unwrap();
    println!("\n=== Synthesized ===\n{}", combined.trim());
}
```



Pattern 17 — CONFIG Node: Parameterized Pipeline

```
topic=Rust lifetimes and borrow checker
style=precise and example-driven
max_words=60
```

```rust
fn main() { println!("Rust lifetimes and borrow checker"); }
```

```rust
fn main() { println!("precise and example-driven"); }
```

```rust
use std::{env, fs};
fn main() {
    let topic   =
fs::read_to_string(env::var("MSH_VAR_topic").unwrap()).unwrap();
    let style   =
fs::read_to_string(env::var("MSH_VAR_style").unwrap()).unwrap();
    let exp     =
fs::read_to_string(env::var("MSH_VAR_explanation").unwrap()).unwrap();
    let kw      =
fs::read_to_string(env::var("MSH_VAR_keywords").unwrap()).unwrap();
    let sep = "\u{2550}".repeat(50);
    println!("{}\n{:^50}\n{}", sep, "Rust Pipeline Report", sep);
    println!("Topic    : {}", topic.trim());
    println!("Style    : {}", style.trim());
    println!("{}", "\u{2500}".repeat(50));
    println!("Explanation:\n{}", exp.trim());
    println!("{}", "\u{2500}".repeat(50));
    println!("Keywords: {}", kw.trim());
    println!("{}", sep);
}
```

Pattern 18 — FOREACH + Async LLM: Parallel Batch Processing

```
fn main() {
    print!("Builder\nNewtype\nTypeState\nRaiiGuard");
}

for p in "$(printenv MSH_VAR_explanation)" "$(printenv MSH_VAR_analogy)"; do
    i=0
    while [ ! -s "$p" ] && [ $i -lt 180 ]; do sleep 1; i=$((i+1)); done
    if [ ! -s "$p" ]; then echo "[no response]" > "$p"; fi
done

echo
"\u250c\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\
u2500\u2500\u2500\u2500\u2510"
printf "\u2502 Pattern: %-44s \u2502\n" "$(cat $(printenv MSH_VAR_pattern))"
echo
"\u251c\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\
u2500\u2500\u2500\u2500\u2524"
printf "\u2502 \U0001f4a1 %-50s \u2502\n" "$(cat $(printenv
MSH_VAR_explanation))"
printf "\u2502 \U0001f517 %-50s \u2502\n" "$(cat $(printenv
MSH_VAR_analogy))"
echo
"\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
```

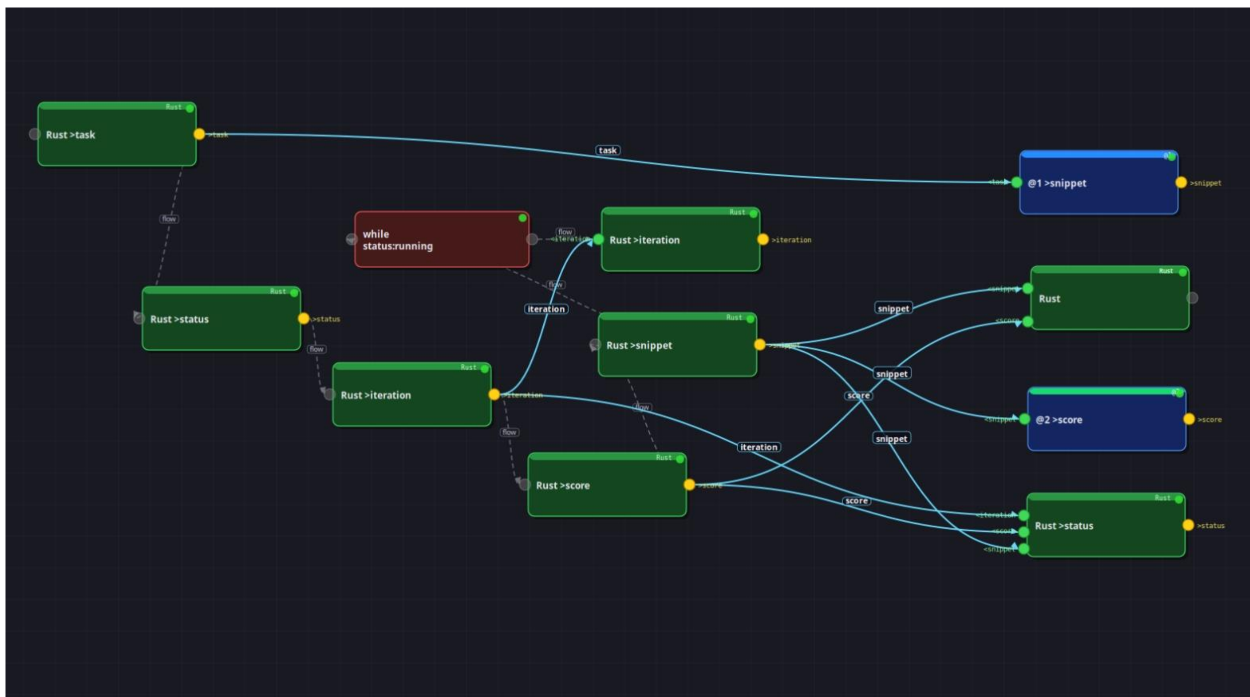
```
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\
u2500\u2500\u2500\u2500\u2518"
echo ""

echo "=== Batch complete: 4 Rust design patterns described ==="
```

Pattern 19 — WHILE Quality Gate: Generate Until Threshold

```rust
fn main() {
    println!("Write a Rust iterator chain: take Vec<i32>, filter evens, \
        square them, collect into Vec<i32>. One expression, no fn main().");
}

fn main() { println!("running"); }

fn main() { println!("0"); }

fn main() { println!("0"); }

fn main() { println!(""); }

use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_iteration").unwrap();
    let val: u64 = fs::read_to_string(&path).unwrap_or_else(|_| "0".into())
        .trim().parse().unwrap_or(0);
    let new_val = val + 1;
    fs::write(&path, format!("{}", new_val)).unwrap();
    print!("{}", new_val);
}
```

```rust
use std::{env, fs};
fn main() {
    let iter    =
fs::read_to_string(env::var("MSH_VAR_iteration").unwrap()).unwrap();
    let score   =
fs::read_to_string(env::var("MSH_VAR_score").unwrap()).unwrap();
    let snippet =
fs::read_to_string(env::var("MSH_VAR_snippet").unwrap()).unwrap();
    let status_path = env::var("MSH_VAR_status").unwrap();
    let score_n: u64 = score.trim().parse().unwrap_or(0);
    println!("[Iter {}] Score={}/10  Snippet: {}", iter.trim(), score_n,
snippet.trim());
    fs::write(&status_path, if score_n >= 8 { "done" } else { "running"
}).unwrap();
}
```

```rust
use std::{env, fs};
fn main() {
    let snippet =
fs::read_to_string(env::var("MSH_VAR_snippet").unwrap()).unwrap();
    let score   =
fs::read_to_string(env::var("MSH_VAR_score").unwrap()).unwrap();
    println!("=== Accepted (score={}/10) ===\n{}", score.trim(),
snippet.trim());
}
```

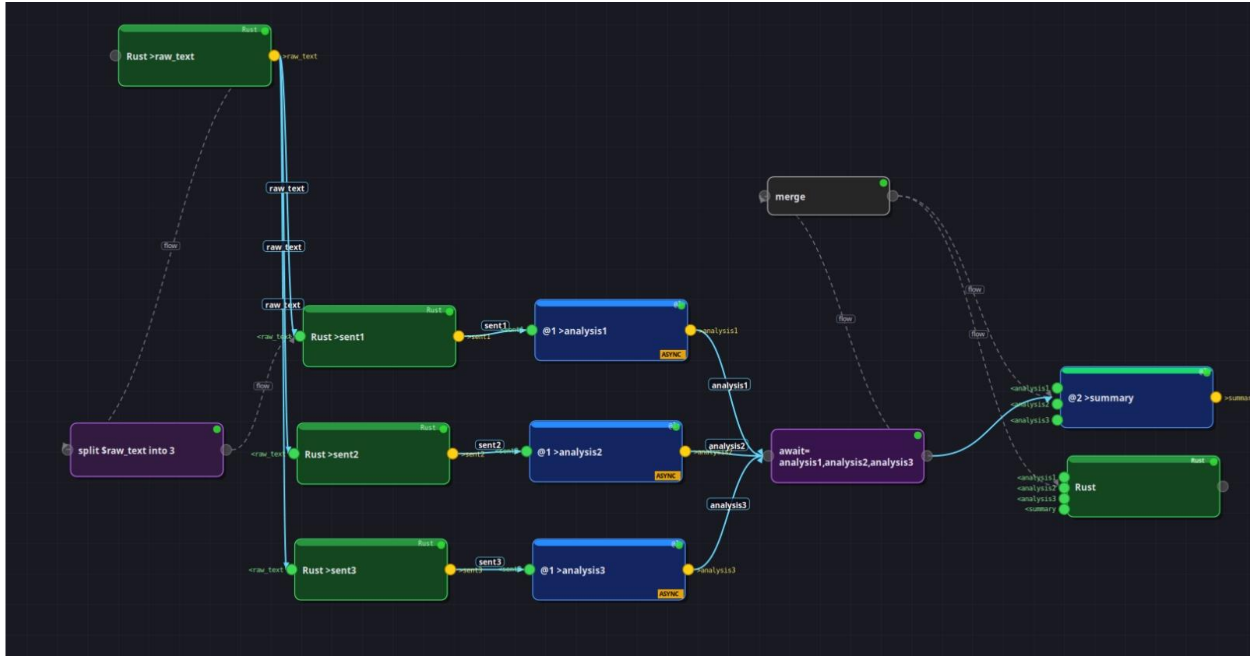


Pattern 20 — SPLIT + Async + MERGE: Map-Reduce Pipeline

```rust
fn main() {
    println!("Rust guarantees memory safety without garbage collection via
```

```
ownership. \
        The borrow checker enforces references never outlive their data. \
        Stack allocation is preferred; heap uses Box, Rc, or Arc smart
pointers. \
        The Drop trait ensures deterministic resource cleanup at end of
scope.");
}

use std::{env, fs};
fn main() {
    let raw =
fs::read_to_string(env::var("MSH_VAR_raw_text").unwrap()).unwrap();
    let s: Vec<&str> = raw.trim().split('.').map(|s| s.trim()).filter(|s|
!s.is_empty()).collect();
    println!("{}", s.get(0).unwrap_or(&""));
}

use std::{env, fs};
fn main() {
    let raw =
fs::read_to_string(env::var("MSH_VAR_raw_text").unwrap()).unwrap();
    let s: Vec<&str> = raw.trim().split('.').map(|s| s.trim()).filter(|s|
!s.is_empty()).collect();
    println!("{}",
s.iter().skip(1).take(2).cloned().collect::<Vec<_>>().join(". "));
}

use std::{env, fs};
fn main() {
    let raw =
fs::read_to_string(env::var("MSH_VAR_raw_text").unwrap()).unwrap();
    let s: Vec<&str> = raw.trim().split('.').map(|s| s.trim()).filter(|s|
!s.is_empty()).collect();
    println!("{}", s.iter().skip(3).cloned().collect::<Vec<_>>().join(". "));
}

for p in "$(printenv MSH_VAR_analysis1)" "$(printenv MSH_VAR_analysis2)"
"$(printenv MSH_VAR_analysis3)"; do
    i=0; while [ ! -s "$p" ] && [ $i -lt 180 ]; do sleep 1; i=$((i+1)); done
    if [ ! -s "$p" ]; then echo "[no response]" > "$p"; fi
done

use std::{env, fs};
fn main() {
    println!("=== Map ===");
    for (label, var) in &[("Chunk 1","analysis1"),("Chunk
2","analysis2"),("Chunk 3","analysis3")] {
        let c = fs::read_to_string(env::var(&format!("MSH_VAR_{}",
var)).unwrap()).unwrap();
        println!("  {}: {}", label, c.trim());
    }
```

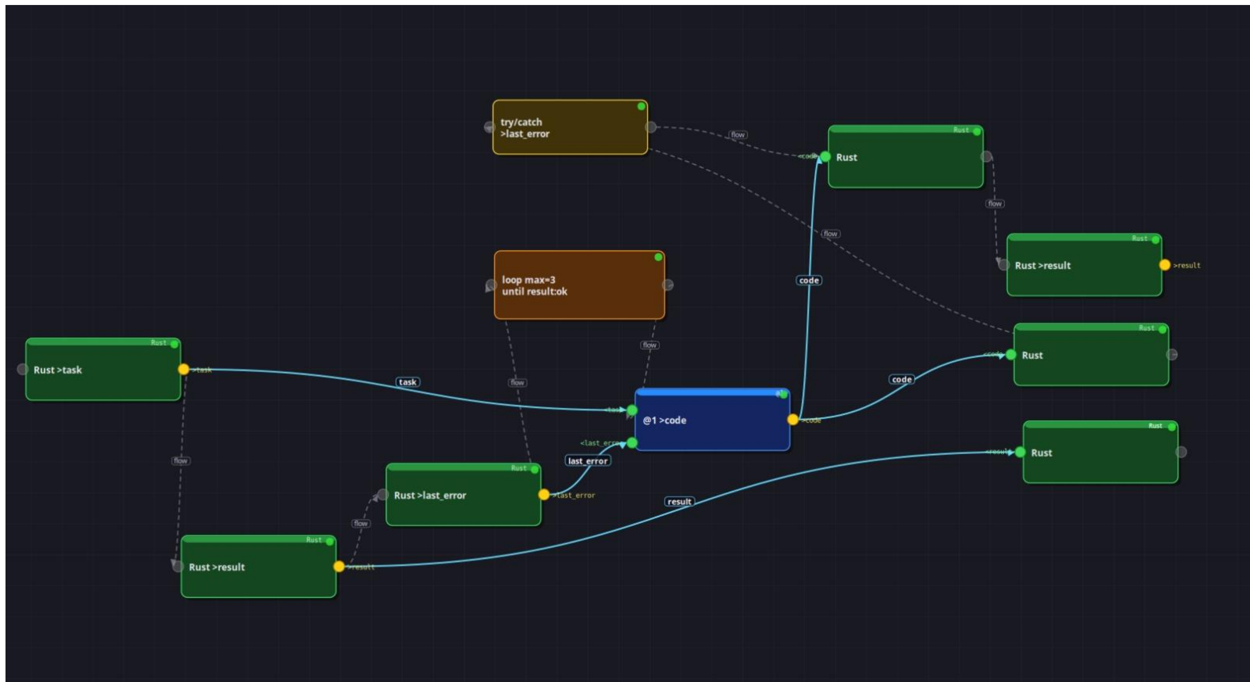
```
    let summary =
fs::read_to_string(env::var("MSH_VAR_summary").unwrap()).unwrap();
    println!("\n=== Reduce ===\n{}", summary.trim());
}
```



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

```
fn main() {
    println!("Write Rust fn median(data: &mut Vec<f64>) -> Option<f64> \
        that returns median (sorts in place), None for empty. \
        Include fn main() testing with [3.0,1.0,4.0,1.0,5.0] and empty
vec.");
}

fn main() { println!("fail"); }

fn main() { println!("none"); }

use std::{env, fs};
fn main() {
    println!("=== Generated ===\n{}\n",

fs::read_to_string(env::var("MSH_VAR_code").unwrap()).unwrap().trim());
}

use std::{env, fs, process::Command};
fn main() {
    let code_path   = env::var("MSH_VAR_code").unwrap();
    let result_path = env::var("MSH_VAR_result").unwrap();
    let err_path    = "/tmp/msh_p21_err.txt";
    let src         = "/tmp/msh_p21_gen.rs";
    let bin         = "/tmp/msh_p21_bin";
```

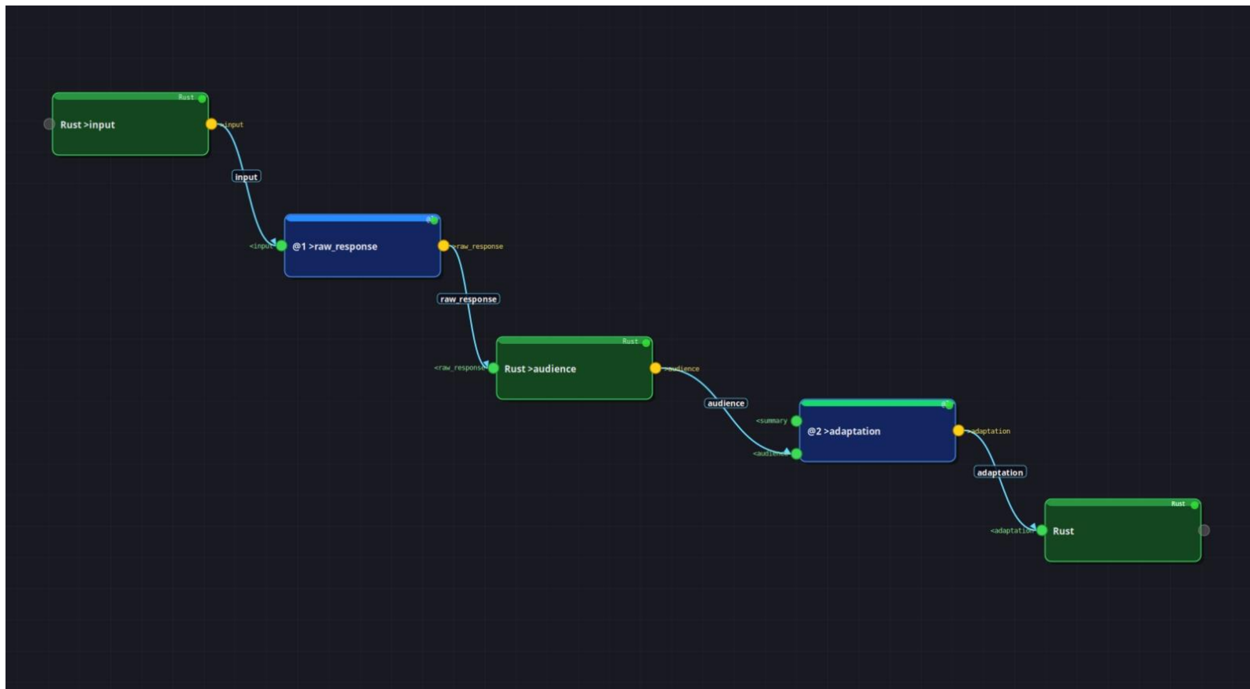
```
    fs::copy(&code_path, src).unwrap();

    let c = Command::new("rustc").args([src, "-o", bin]).output().unwrap();
    if !c.status.success() {
        // write error to tmp file so CATCH can pass it to LLM
        let err = String::from_utf8_lossy(&c.stderr).to_string();
        fs::write(err_path, &err).unwrap();
        eprintln!("{}", err);
        std::process::exit(1);
    }
    let run = Command::new(bin).output().unwrap();
    print!("{}", String::from_utf8_lossy(&run.stdout));
    fs::write(&result_path, "ok\n").unwrap();
    println!("ok");
}

use std::{env, fs};
fn main() {
    // pass compile error to last_error so LLM can fix it next iteration
    let err = fs::read_to_string("/tmp/msh_p21_err.txt")
        .unwrap_or_else(|_| "unknown error".into());
    fs::write(env::var("MSH_VAR_last_error").unwrap(), &err).unwrap();
    fs::write(env::var("MSH_VAR_result").unwrap(), "fail\n").unwrap();
    println!("fail");
}

use std::{env, fs};
fn main() {
    println!("=== Final: {} ===",

fs::read_to_string(env::var("MSH_VAR_result").unwrap()).unwrap().trim());
}
```



Pattern 22 — Multi-Variable Output: Structured Field Extraction

```
fn main() {
    println!("HashMap::entry(&mut self, key: K) -> Entry<'_, K, V> provides \
        in-place mutable access to an entry for key-based insertion or
modification, \
        avoiding double-lookups via the Entry enum with occupied and vacant
variants.");
}

use std::{env, fs};
fn extract<'a>(text: &'a str, prefix: &str) -> &'a str {
    text.lines().find(|l| l.starts_with(prefix))
        .and_then(|l| l.strip_prefix(prefix))
        .map(|s| s.trim()).unwrap_or("n/a")
}
fn main() {
    let text =
fs::read_to_string(env::var("MSH_VAR_raw_response").unwrap()).unwrap();
    let summary   = extract(&text, "SUMMARY: ");
    let signature = extract(&text, "SIGNATURE: ");
    let audience  = extract(&text, "AUDIENCE: ");
    fs::write(env::var("MSH_VAR_summary").unwrap(),   summary).unwrap();
    fs::write(env::var("MSH_VAR_signature").unwrap(), signature).unwrap();
    fs::write(env::var("MSH_VAR_audience").unwrap(),  audience).unwrap();
    println!("Summary   : {}", summary);
    println!("Signature : {}", signature);
    println!("Audience  : {}", audience);
}
```

```rust
use std::{env, fs};
fn main() {
    let c =
fs::read_to_string(env::var("MSH_VAR_adaptation").unwrap()).unwrap();
    println!("\n=== Adapted for Audience ===\n{}", c.trim());
}
```



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

```
subject=Rust async/await and the Future trait
target_audience=software engineer familiar with threads but new to async
quality_threshold=7
```

```rust
fn main() { println!("Rust async/await and the Future trait"); }
```

```rust
fn main() { println!("software engineer familiar with threads but new to
async"); }
```

```rust
fn main() { println!("running"); }
```

```rust
fn main() { println!("0"); }
```

```rust
fn main() { println!("0"); }
```

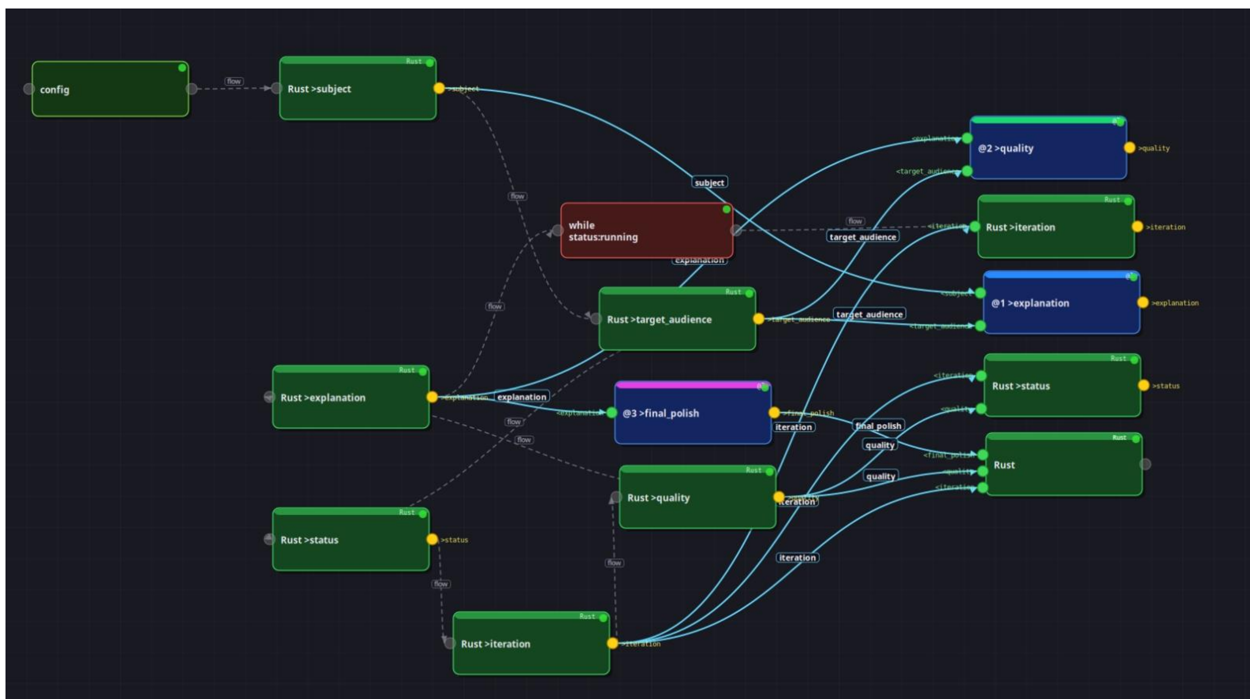
```rust
fn main() { println!(""); }
```

```rust
use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_iteration").unwrap();
    let val: u64 = fs::read_to_string(&path).unwrap_or_else(|_| "0".into())
        .trim().parse().unwrap_or(0);
```

```
    let new_val = val + 1;
    fs::write(&path, format!("{}", new_val)).unwrap();
    print!("{}", new_val);
}

use std::{env, fs};
fn main() {
    let iter =
fs::read_to_string(env::var("MSH_VAR_iteration").unwrap()).unwrap();
    let q    =
fs::read_to_string(env::var("MSH_VAR_quality").unwrap()).unwrap();
    let sp   = env::var("MSH_VAR_status").unwrap();
    let qn: u64 = q.trim().parse().unwrap_or(0);
    println!("[Iter {}] Quality: {}/10", iter.trim(), qn);
    fs::write(&sp, if qn >= 7 { "done" } else { "running" }).unwrap();
}

use std::{env, fs};
fn main() {
    let polish  =
fs::read_to_string(env::var("MSH_VAR_final_polish").unwrap()).unwrap();
    let quality =
fs::read_to_string(env::var("MSH_VAR_quality").unwrap()).unwrap();
    let iter    =
fs::read_to_string(env::var("MSH_VAR_iteration").unwrap()).unwrap();
    println!("=== Final (score={}/10, iterations={}) ===", quality.trim(),
iter.trim());
    println!("{}", polish.trim());
}
```



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

```
fn main() {
    let s1 = r#"fn double(x: i32) -> i32 { x * 2 } fn main() { println!("{}",
double(21)); }"#;
    let s2 = r#"fn main() { let v: Vec<i32> = (1..=5).map(|x| x*x).collect();
println!("{:?}", v); }"#;
    let s3 = r#"fn main() { let s = String::from("Ferris"); println!("Hello,
{}!", s); }"#;
    let broken = r#"fn main( { let x = ; }"#;
    print!("{}\n{}\n{}\n{}", s1, s2, broken, s3);
}

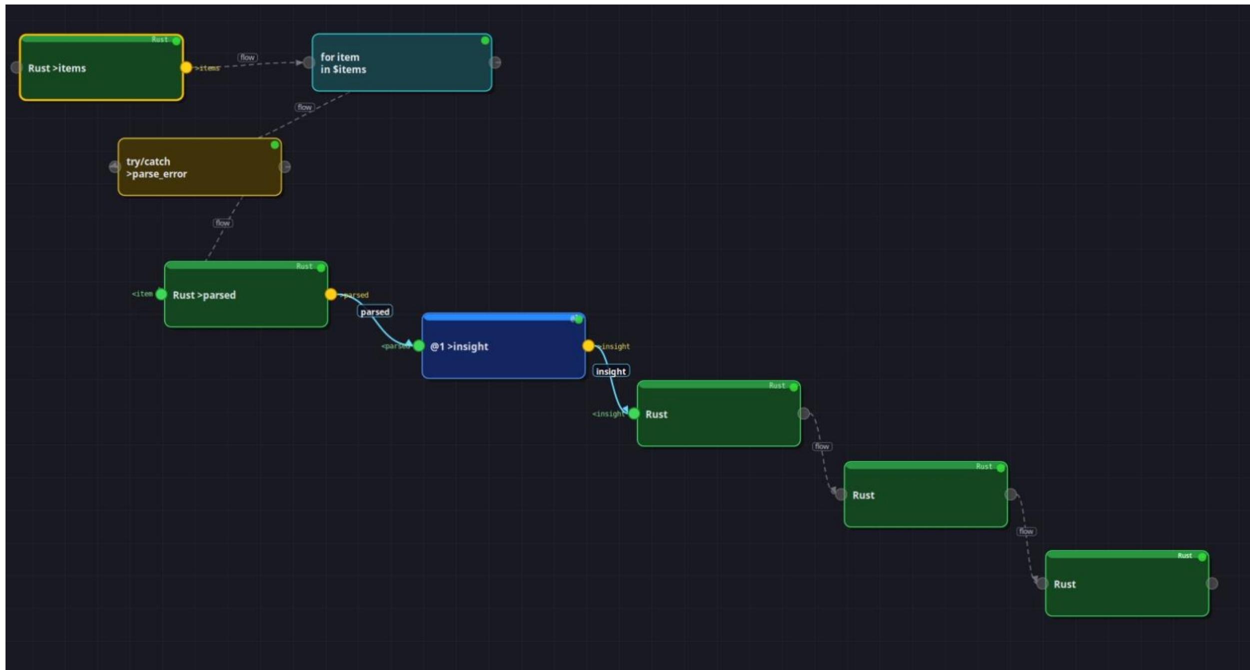
use std::{env, fs, process::Command};
fn main() {
    let item_path   = env::var("MSH_VAR_item").unwrap();
    let parsed_path = env::var("MSH_VAR_parsed").unwrap();
    let snippet = fs::read_to_string(&item_path).unwrap();
    let src = "/tmp/mshell_p24_snippet.rs";
    let bin = "/tmp/mshell_p24_snippet_bin";
    fs::write(src, snippet.trim()).unwrap();
    let c = Command::new("rustc").args(["--edition","2021",src,"-
o",bin]).output().unwrap();
    if !c.status.success() {
        eprintln!("{}", String::from_utf8_lossy(&c.stderr));
        std::process::exit(1);
    }
    fs::write(&parsed_path, snippet.trim()).unwrap();

}

use std::{env, fs};
fn main() {
    let c =
fs::read_to_string(env::var("MSH_VAR_insight").unwrap()).unwrap();
    println!("[OK] {}", c.trim());
}

fn main() {
    println!("[ERR] Compilation failed: try_block_failed");
}

fn main() {
    println!("=== Batch complete. Errors isolated, pipeline never stopped.
===");
}
```



Quick Reference: Common Rust Patterns

Read entire file to string:

```
let text = fs::read_to_string(env::var("MSH_VAR_name").unwrap()).unwrap();
```

Read multiple variables:

```
fn slurp(key: &str) -> String {
    env::var(key).ok()
        .and_then(|p| fs::read_to_string(&p).ok())
        .unwrap_or_else(|| "[not available]".to_string())
}
```

WHILE exit (write status to file):

```
fs::write(env::var("MSH_VAR_status").unwrap(),
    if done { "done\n" } else { "running\n" }).unwrap();
```

LOOP exit (print as last stdout line):

```
// Last line of last block in loop body
println!("{}", if success { "ok" } else { "fail" });
```

Appendix I: Code examples for each pattern

/home/igor > Sent to mshell (3966 bytes)

Received from GUI editor:

Pattern 1 Linear Data Pipeline (Rust All Stages)

````
```rust >primes
fn sieve(limit: usize) -> Vec<usize> {
    let mut is_prime = vec![true; limit + 1];
    is_prime[0] = false;
    if limit > 0 { is_prime[1] = false; }
    let mut i = 2;
    while i * i <= limit {
        if is_prime[i] {
            let mut j = i * i;
            while j <= limit { is_prime[j] = false; j += i; }
        }
        i += 1;
    }
    is_prime.iter().enumerate().filter(|(_, &p)| p).map(|(n, _)| n).collect()
}
fn main() {
    let first20: Vec<usize> = sieve(80).into_iter().take(20).collect();
    let out: Vec<String> = first20.iter().map(|n| n.to_string()).collect();
    println!("{}", out.join(","));
}
```
```rust <primes >stats
````
````

```
use std::{env, fs};

fn main() {
    let path = env::var("MSH_VAR_primes").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let nums: Vec<f64> = raw.trim().split(',').map(|s| s.parse().unwrap()).collect();
    let n = nums.len() as f64;
    let sum: f64 = nums.iter().sum();
    let mean = sum / n;
    let variance = nums.iter().map(|x| (x - mean).powi(2)).sum::<f64>() / n;
    let mut sorted = nums.clone();
    sorted.sort_by(|a, b| a.partial_cmp(b).unwrap());
    let median = if sorted.len() % 2 == 0 {
        (sorted[sorted.len()/2 - 1] + sorted[sorted.len()/2]) / 2.0
    } else { sorted[sorted.len()/2] };
    println!("count={} sum={} mean={:.2} variance={:.2} median={} min={} max={}",
        nums.len(), sum as usize, mean, variance,
        median as usize, sorted[0] as usize, sorted[sorted.len()-1] as usize);
}
```

```rust <stats >table
use std::{env, fs};

fn main() {
    let path = env::var("MSH_VAR_stats").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
println!("\u{250c}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{25
00}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u
{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500
```

```
}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2510}");
   println!("\u{2502}        Prime Numbers \u{2014} Statistics      \u{2502}");
println!("\u{251c}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{252c}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2524}");
   for part in raw.trim().split_whitespace() {
      if let Some((k, v)) = part.split_once('=') {
         println!("\u{2502} {:12} \u{2502} {:>24} \u{2502}", k, v);
      }
   }
println!("\u{2514}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2534}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2500}\u{2518}");
}
```

```rust <primes >histogram
use std::{env, fs};
fn main() {
   let path = env::var("MSH_VAR_primes").unwrap();
   let raw = fs::read_to_string(&path).unwrap();
   let nums: Vec<usize> = raw.trim().split(',').map(|s| s.parse().unwrap()).collect();
   let mut freq = [0usize; 10];
   for &n in &nums { freq[n % 10] += 1; }
   println!("Last-digit frequency histogram:");
   for (digit, &count) in freq.iter().enumerate() {
      if count > 0 { println!("  {} | {}", digit, "\u{2588}".repeat(count * 4)); }
```

```
    }
}
```

```rust <table <histogram
use std::{env, fs};

fn main() {
    for var in &["table", "histogram"] {
        let path = env::var(&format!("MSH_VAR_{}", var)).unwrap();
        println!("{}", fs::read_to_string(&path).unwrap().trim());
        println!();
    }
    println!("=== Pipeline complete ===");
}
```

2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71

count=20 sum=639 mean=31.95 variance=479.55 median=30 min=2 max=71

\u250c\u2500\u2510

\u2502   Prime Numbers \u2014 Statistics   \u2502

\u251c\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u252c\u2500\u2524

\u2502 count   \u2502   20 \u2502

\u2502 sum   \u2502   639 \u2502

\u2502 mean   \u2502   31.95 \u2502

\u2502 variance   \u2502   479.55 \u2502

\u2502 median   \u2502   30 \u2502

\u2502 min   \u2502   2 \u2502

\u2502 max   \u2502   71 \u2502

\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2534\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2518

Last-digit frequency histogram:

  1 |
\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u258
8\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588

  2 | \u2588\u2588\u2588\u2588

  3 |
\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u258
8\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588

  5 | \u2588\u2588\u2588\u2588

  7 |
\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u258
8\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588

  9 |
\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u258
8

/home/igor >
\u250c\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500
\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2510

\u2502   Prime Numbers \u2014 Statistics   \u2502

\u251c\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500
\u2500\u2500\u2500\u252c\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500
\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2524

\u2502 count   \u2502   20 \u2502

\u2502 sum   \u2502   639 \u2502

\u2502 mean   \u2502   31.95 \u2502

\u2502 variance   \u2502   479.55 \u2502

\u2502 median   \u2502   30 \u2502

\u2502 min   \u2502   2 \u2502

\u2502 max   \u2502   71 \u2502

\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2534\u2500\u2518

Last-digit frequency histogram:

1 | \u2588

2 | \u2588\u2588\u2588\u2588

3 | \u2588

5 | \u2588\u2588\u2588\u2588

7 | \u2588

9 | \u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588\u2588

=== Pipeline complete ===

/home/igor > Sent to mshell (2530 bytes)

Received from GUI editor:

Pattern 2 LLM in the Middle (Rust + LLM + Rust)

```rust >bench_data
fn main() {
    let metrics = [
        ("integer_ops_per_sec",  "4_820_000_000"),
        ("float_ops_per_sec",    "2_310_000_000"),
        ("memory_bandwidth_gbs", "48.7"),
        ("cache_miss_rate_pct",  "3.2"),
        ("p50_latency_us",       "12"),
        ("p95_latency_us",       "87"),
        ("p99_latency_us",       "214"),
        ("thread_count",         "16"),
    ];
    println!("=== Benchmark Report ===");
    for (k, v) in &metrics { println!("{}: {}", k, v); }
}
```

<!--@1 <bench_data >analysis

The input contains a system benchmark report.

Analyze the results: identify bottlenecks, highlight strengths, and provide

3 concrete recommendations. Be concise \u2014 3 short paragraphs max.

-->

```rust <analysis
use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_analysis").unwrap();
    let text = fs::read_to_string(&path).unwrap();
```

```
    let text = text.trim();
    let words = text.split_whitespace().count();
    let sentences = text.chars().filter(|&c| matches!(c, '.' | '!' | '?')).count();
    let paras = text.split("\n\n").filter(|s| !s.trim().is_empty()).count();
println!("\u{2554}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{25
50}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u
{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550
}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2
550}\u{2550}\u{2557}");
    println!("\u{2551}     LLM Performance Analysis       \u{2551}");
println!("\u{2560}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{25
50}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u
{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550
}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2
550}\u{2550}\u{2563}");
    println!("\u{2551} Words   : {:>28} \u{2551}", words);
    println!("\u{2551} Sentences: {:>27} \u{2551}", sentences);
    println!("\u{2551} Paragraphs: {:>26} \u{2551}", paras);
println!("\u{255a}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{25
50}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u
{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550
}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2550}\u{2
550}\u{2550}\u{255d}");
    println!("\n{}", text);
}
```

=== Benchmark Report ===

integer_ops_per_sec: 4_820_000_000

float_ops_per_sec: 2_310_000_000

memory_bandwidth_gbs: 48.7

cache_miss_rate_pct: 3.2

p50_latency_us: 12

p95_latency_us: 87

p99_latency_us: 214

thread_count: 16

Performance Analysis: This system shows strong integer performance (4.8B ops/sec) but relatively weak floating-point throughput at only 48% of integer performance, suggesting potential FPU limitations or workload inefficiencies. The excellent cache hit rate (96.8%) and high memory bandwidth (48.7 GB/s) indicate good memory hierarchy performance, while the 16-thread configuration appears well-utilized.

Bottlenecks: The primary constraint is floating-point performance, which may limit compute-intensive applications. The significant latency tail (p99 at 214\u03bcs vs p50 at 12\u03bcs) suggests occasional scheduling delays or resource contention, possibly from thread synchronization or occasional cache pressure.

Recommendations: (1) Profile floating-point workloads for vectorization opportunities using SIMD instructions to boost FP throughput, (2) investigate p99 latency spikes by analyzing thread scheduling patterns and reducing synchronization overhead, and (3) consider workload partitioning to leverage the strong integer performance while offloading FP-heavy tasks to specialized cores or accelerators.

\u2554\u2550\u2557

\u2551 LLM Performance Analysis \u2551

\u2560\u2550\u2563

\u2551 Words : 138 \u2551

\u2551 Sentences: 8 \u2551

\u2551 Paragraphs: 3 \u2551

\u255a\u2550\u255d

Performance Analysis: This system shows strong integer performance (4.8B ops/sec) but relatively weak floating-point throughput at only 48% of integer performance, suggesting potential FPU limitations or workload inefficiencies. The excellent cache hit rate (96.8%) and high memory bandwidth (48.7 GB/s) indicate good memory hierarchy performance, while the 16-thread configuration appears well-utilized.

Bottlenecks: The primary constraint is floating-point performance, which may limit compute-intensive applications. The significant latency tail (p99 at 214\u03bcs vs p50 at 12\u03bcs) suggests occasional scheduling delays or resource contention, possibly from thread synchronization or occasional cache pressure.

Recommendations: (1) Profile floating-point workloads for vectorization opportunities using SIMD instructions to boost FP throughput, (2) investigate p99 latency spikes by analyzing thread scheduling patterns and reducing synchronization overhead, and (3) consider workload partitioning to leverage the strong integer performance while offloading FP-heavy tasks to specialized cores or accelerators.

/home/igor > Sent to mshell (2373 bytes)

Received from GUI editor:

Pattern 3 Fan-Out: One Variable \u2192 Many Consumers

```rust >dataset
fn main() {
    let mut state: u64 = 0xDEAD_BEEF_1234_5678;
    let mut nums = Vec::new();
    for _ in 0..15 {
        state = state.wrapping_mul(6364136223846793005).wrapping_add(1442695040888963407);
        nums.push((state >> 33) % 200 + 2);
    }
    println!("{}", nums.iter().map(|n| n.to_string()).collect::<Vec<_>>().join(" "));
}
```

```rust <dataset >stats
use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_dataset").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let nums: Vec<f64> = raw.trim().split_whitespace().map(|s| s.parse().unwrap()).collect();
    let n = nums.len() as f64;
    let mean = nums.iter().sum::<f64>() / n;
    let std_dev = (nums.iter().map(|x| (x - mean).powi(2)).sum::<f64>() / n).sqrt();
    let mut sorted = nums.clone();
    sorted.sort_by(|a, b| a.partial_cmp(b).unwrap());
    println!("n={} mean={:.1} std_dev={:.1} min={} max={} median={}",
        nums.len(), mean, std_dev,
        sorted[0] as usize, sorted[sorted.len()-1] as usize,
        sorted[sorted.len()/2] as usize);
}
```

```rust <dataset >primes_in_set
use std::{env, fs};
fn is_prime(n: u64) -> bool {
    if n < 2 { return false; }
    if n == 2 { return true; }
    if n % 2 == 0 { return false; }
    let mut i = 3u64;
    while i * i <= n { if n % i == 0 { return false; } i += 2; }
    true
}
```

```
fn main() {
    let path = env::var("MSH_VAR_dataset").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let primes: Vec<u64> = raw.trim().split_whitespace()
        .map(|s| s.parse().unwrap()).filter(|&n| is_prime(n)).collect();
    println!("Primes found ({}): {}", primes.len(),
        primes.iter().map(|n| n.to_string()).collect::<Vec<_>>().join(", "));
}
```

<!--@1 <dataset

The input contains a space-separated list of integers.

In exactly two sentences describe what kind of dataset this looks like and

whether it appears random, clustered, or has a notable pattern.

-->

```rust <stats <primes_in_set
use std::{env, fs};
fn main() {
    for (label, var) in &[("Statistics", "stats"), ("Primes in set", "primes_in_set")] {
        let path = env::var(&format!("MSH_VAR_{}", var)).unwrap();
        println!("--- {} ---\n{}\n", label, fs::read_to_string(&path).unwrap().trim());
    }
}
```

----

29 62 132 19 48 104 40 35 100 140 56 61 56 90 99

n=15 mean=71.4 std_dev=36.0 min=19 max=140 median=61

Primes found (3): 29, 19, 61

This dataset appears to be a collection of moderate-sized integers ranging from 19 to 140 with no immediately obvious mathematical pattern or sequence, suggesting it could represent measurements, scores, or operational data from some real-world system. The values appear somewhat clustered in different ranges (19-48, 56-104, 132-140) rather than being uniformly random, indicating possible groupings or categories within the underlying data source.

/home/igor > --- Statistics ---

n=15 mean=71.4 std_dev=36.0 min=19 max=140 median=61

--- Primes in set ---

Primes found (3): 29, 19, 61

/home/igor > Sent to mshell (1151 bytes)

Received from GUI editor:

Pattern 4 LLM Code Generation \u2192 Execute via Variable

```rust >task
fn main() {
    println!("Write a Rust program that generates Fibonacci numbers up to 1000, \
        filters only those divisible by 3, and prints each with its index and running sum.");
}
```

<!--@1 <task >code

Write ONLY pure Rust code. No markdown fences, no explanation.

Must include fn main() and compile with rustc stable.

-->

```rust <code
use std::{env, fs, process::Command};

fn main() {
```

```
    let code_path = env::var("MSH_VAR_code").unwrap();

    let source = fs::read_to_string(&code_path).unwrap();

    println!("=== Generated Rust Code ===\n{}\n", source.trim());

    let src = "/tmp/mshell_generated.rs";

    let bin = "/tmp/mshell_generated_bin";

    fs::write(src, &source).unwrap();

    let compile = Command::new("rustc").args([src, "-o", bin]).output().unwrap();

    if !compile.status.success() {

        eprintln!("{}", String::from_utf8_lossy(&compile.stderr));

        return;

    }

    println!("=== Output ===");

    print!("{}", String::from_utf8_lossy(&Command::new(bin).output().unwrap().stdout));

}
```

Write a Rust program that generates Fibonacci numbers up to 1000, filters only those divisible by 3, and prints each with its index and running sum.

```
fn main() {

    let mut a = 0;

    let mut b = 1;

    let mut fib_index = 0;

    let mut filtered_index = 0;

    let mut running_sum = 0;

    while a <= 1000 {

        if a % 3 == 0 {

            running_sum += a;
```

```
            println!("Index: {}, Fib[{}]: {}, Running Sum: {}", filtered_index, fib_index, a,
running_sum);
            filtered_index += 1;
        }
        let next = a + b;
        a = b;
        b = next;
        fib_index += 1;
    }
}
/home/igor > === Generated Rust Code ===
fn main() {
    let mut a = 0;
    let mut b = 1;
    let mut fib_index = 0;
    let mut filtered_index = 0;
    let mut running_sum = 0;

    while a <= 1000 {
        if a % 3 == 0 {
            running_sum += a;
            println!("Index: {}, Fib[{}]: {}, Running Sum: {}", filtered_index, fib_index, a,
running_sum);
            filtered_index += 1;
        }
        let next = a + b;
        a = b;
        b = next;
```

~~~~
        fib_index += 1;
    }
}
=== Output ===
Index: 0, Fib[0]: 0, Running Sum: 0
Index: 1, Fib[4]: 3, Running Sum: 3
Index: 2, Fib[8]: 21, Running Sum: 24
Index: 3, Fib[12]: 144, Running Sum: 168
Index: 4, Fib[16]: 987, Running Sum: 1155
----------------------------------------------------------------------
/home/igor > Sent to mshell (1887 bytes)
Received from GUI editor:
--------------------------------------
# Pattern 5  Two-LLM Review Chain
--------------------------------------
```rust >task
fn main() {
    println!("Write a Rust function that counts word frequencies in a string \
        using HashMap, returns top-3 most frequent words sorted by frequency desc \
        then alphabetically. Include fn main() with a demo.");
}
```
<!--@1 <task >code
Write only Rust code, no markdown fences. Must compile with rustc (stable).
Include fn main() with a demo.
-->
```rust <code
~~~~

````
use std::{env, fs};

fn main() {

    let path = env::var("MSH_VAR_code").unwrap();

    println!("=== Model @1 Generated ===\n{}\n",

        fs::read_to_string(&path).unwrap().trim());

}

```


<!--@2 <code >review

Review this Rust code for: algorithm correctness, ownership correctness,

idiomatic Rust (iterators, error handling), edge cases.

Reply in 3 bullet points max. Be specific.

-->

```rust <review

use std::{env, fs};

fn main() {

    let path = env::var("MSH_VAR_review").unwrap();

    println!("=== Model @2 Review ===\n{}\n",

        fs::read_to_string(&path).unwrap().trim());

}

```

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

Improve the Rust code based on the review.

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

-->

```rust <improved

use std::{env, fs, process::Command};
````

```
fn main() {
    let code_path = env::var("MSH_VAR_improved").unwrap();
    let source = fs::read_to_string(&code_path).unwrap();
    println!("=== Final Improved Code ===\n{}\n", source.trim());
    let src = "/tmp/mshell_improved.rs";
    let bin = "/tmp/mshell_improved_bin";
    fs::write(src, &source).unwrap();
    let c = Command::new("rustc").args([src, "-o", bin]).output().unwrap();
    if !c.status.success() { eprintln!("{}", String::from_utf8_lossy(&c.stderr)); return; }
    println!("=== Executing ===");
    print!("{}", String::from_utf8_lossy(&Command::new(bin).output().unwrap().stdout));
}
```

Write a Rust function that counts word frequencies in a string using HashMap, returns top-3 most frequent words sorted by frequency desc then alphabetically. Include fn main() with a demo.

```
use std::collections::HashMap;

fn top_words(text: &str) -> Vec<(String, usize)> {
    let mut word_counts = HashMap::new();
    for word in text.split_whitespace() {
        let clean_word = word.to_lowercase()
            .chars()
            .filter(|c| c.is_alphabetic())
            .collect::<String>();
        if !clean_word.is_empty() {
            *word_counts.entry(clean_word).or_insert(0) += 1;
        }
```

```
    }
    let mut sorted_words: Vec<(String, usize)> = word_counts.into_iter().collect();
    sorted_words.sort_by(|a, b| {
        b.1.cmp(&a.1).then_with(|| a.0.cmp(&b.0))
    });
    sorted_words.into_iter().take(3).collect()
}
fn main() {
    let text = "the quick brown fox jumps over the lazy dog the fox is quick and the dog is
lazy";
    let result = top_words(text);
    println!("Top 3 most frequent words:");
    for (word, count) in result {
        println!("{}: {}", word, count);
    }
}
=== Model @1 Generated ===
use std::collections::HashMap;
fn top_words(text: &str) -> Vec<(String, usize)> {
    let mut word_counts = HashMap::new();
    for word in text.split_whitespace() {
        let clean_word = word.to_lowercase()
            .chars()
            .filter(|c| c.is_alphabetic())
            .collect::<String>();
        if !clean_word.is_empty() {
            *word_counts.entry(clean_word).or_insert(0) += 1;
```

```
    }
  }
  let mut sorted_words: Vec<(String, usize)> = word_counts.into_iter().collect();
  sorted_words.sort_by(|a, b| {
    b.1.cmp(&a.1).then_with(|| a.0.cmp(&b.0))
  });
  sorted_words.into_iter().take(3).collect()
}
fn main() {
  let text = "the quick brown fox jumps over the lazy dog the fox is quick and the dog is lazy";
  let result = top_words(text);
  println!("Top 3 most frequent words:");
  for (word, count) in result {
    println!("{}: {}", word, count);
  }
}
```

- Algorithmically correct for counting and ranking words: it lowercases, strips non-alphabetic chars, counts frequencies, then sorts by descending count and ascending lexicographic order, returning the top 3; behavior on ties is deterministic and matches the sort criteria.

- Ownership and borrowing are correct and efficient enough here: `clean_word` is moved into the `HashMap` and later into the `Vec`, and finally into the result, avoiding unnecessary cloning; there are no lifetime or borrow checker issues.

- Idiomatic use of iterators and `HashMap::entry` is good, but you could improve edge cases by handling empty input explicitly (e.g., early return of an empty `Vec`), and possibly factoring out the word \u201ccleaning\u201d logic or using a more precise tokenizer if punctuation/Unicode behavior matters.

=== Model @2 Review ===

- Algorithmically correct for counting and ranking words: it lowercases, strips non-alphabetic chars, counts frequencies, then sorts by descending count and ascending

lexicographic order, returning the top 3; behavior on ties is deterministic and matches the sort criteria.

- Ownership and borrowing are correct and efficient enough here: `clean_word` is moved into the `HashMap` and later into the `Vec`, and finally into the result, avoiding unnecessary cloning; there are no lifetime or borrow checker issues.

- Idiomatic use of iterators and `HashMap::entry` is good, but you could improve edge cases by handling empty input explicitly (e.g., early return of an empty `Vec`), and possibly factoring out the word \u201ccleaning\u201d logic or using a more precise tokenizer if punctuation/Unicode behavior matters.

```
use std::collections::HashMap;

fn clean_word(word: &str) -> Option<String> {
    let cleaned: String = word.to_lowercase()
        .chars()
        .filter(|c| c.is_alphabetic())
        .collect();


    if cleaned.is_empty() {
        None
    } else {
        Some(cleaned)
    }
}

fn top_words(text: &str) -> Vec<(String, usize)> {
    if text.trim().is_empty() {
        return Vec::new();
    }
    let mut word_counts = HashMap::new();
    for word in text.split_whitespace() {
        if let Some(clean_word) = clean_word(word) {
```

```
        *word_counts.entry(clean_word).or_insert(0) += 1;
        }
    }
    let mut sorted_words: Vec<(String, usize)> = word_counts.into_iter().collect();
    sorted_words.sort_by(|a, b| {
        b.1.cmp(&a.1).then_with(|| a.0.cmp(&b.0))
    });
    sorted_words.into_iter().take(3).collect()
}
fn main() {
    let text = "the quick brown fox jumps over the lazy dog the fox is quick and the dog is
lazy";
    let result = top_words(text);
    println!("Top 3 most frequent words:");
    for (word, count) in result {
        println!("{}: {}", word, count);
    }
    // Test edge cases
    println!("\nTesting empty input:");
    let empty_result = top_words("");
    println!("Empty result: {:?}", empty_result);
}
/home/igor > === Final Improved Code ===
use std::collections::HashMap;
fn clean_word(word: &str) -> Option<String> {
    let cleaned: String = word.to_lowercase()
        .chars()
```

```
        .filter(|c| c.is_alphabetic())
        .collect();
    if cleaned.is_empty() {
        None
    } else {
        Some(cleaned)
    }
}
fn top_words(text: &str) -> Vec<(String, usize)> {
    if text.trim().is_empty() {
        return Vec::new();
    }
    let mut word_counts = HashMap::new();
    for word in text.split_whitespace() {
        if let Some(clean_word) = clean_word(word) {
            *word_counts.entry(clean_word).or_insert(0) += 1;
        }
    }
    let mut sorted_words: Vec<(String, usize)> = word_counts.into_iter().collect();
    sorted_words.sort_by(|a, b| {
        b.1.cmp(&a.1).then_with(|| a.0.cmp(&b.0))
    });
    sorted_words.into_iter().take(3).collect()
}
fn main() {
    let text = "the quick brown fox jumps over the lazy dog the fox is quick and the dog is lazy";
```

```
    let result = top_words(text);
    println!("Top 3 most frequent words:");
    for (word, count) in result {
        println!("{}: {}", word, count);
    }
    // Test edge cases
    println!("\nTesting empty input:");
    let empty_result = top_words("");
    println!("Empty result: {:?}", empty_result);
}
```

=== Executing ===

Top 3 most frequent words:

the: 4

dog: 2

fox: 2

Testing empty input:

Empty result: []

/home/igor > Sent to mshell (1216 bytes)

Received from GUI editor:

Pattern 6 Parallel 3-Model Query

```rust >question
fn main() {
    println!("In Rust, for a high-throughput network service, what is the \
        most important architectural decision regarding async runtime and \
```

````
    concurrency model? Answer in exactly two sentences.");
}
```
<!--@1 <question >ans1
Answer in exactly two sentences. Focus on practical production experience.
-->
<!--@2 <question >ans2
Answer in exactly two sentences. Focus on performance characteristics.
-->
<!--@3 <question >ans3
Answer in exactly two sentences. Focus on maintainability and team productivity.
-->

```rust <ans1 <ans2 <ans3
use std::{env, fs};
fn main() {
  for (label, var) in &[
    ("@1 Production Focus", "ans1"),
    ("@2 Performance Focus", "ans2"),
    ("@3 Maintainability Focus", "ans3"),
  ] {
    let path = env::var(&format!("MSH_VAR_{}", var)).unwrap();
    let content = fs::read_to_string(&path).unwrap();
    println!("\u{250c}\u{2500} {} {}", label,
"\u{2500}".repeat(44usize.saturating_sub(label.len())));
    println!("\u{2502} {}", content.trim());
    println!("\u{2514}{}\n", "\u{2500}".repeat(50));
````

```
   }
}
```

In Rust, for a high-throughput network service, what is the most important architectural decision regarding async runtime and concurrency model? Answer in exactly two sentences.

The most critical decision is choosing between single-threaded async (tokio with `current_thread`) versus multi-threaded work-stealing (tokio with `multi_thread`) based on your workload characteristics - use single-threaded for pure I/O-bound services to avoid context switching overhead, and multi-threaded for mixed I/O/CPU workloads. Equally important is designing your task spawning strategy to avoid blocking the async runtime with CPU-intensive work, using dedicated thread pools via `spawn_blocking` or `rayon` for compute-heavy operations to maintain low latency for network I/O.

Choosing whether to use a single shared multi-threaded async runtime (e.g., Tokio\u2019s multi-threaded scheduler) versus multiple runtimes or a custom event loop is the most important decision, because it directly affects context-switch overhead, cache locality, lock contention, and scheduler fairness under load. Closely coupled to this is deciding how to partition work between lightweight async tasks and dedicated threads for CPU-bound operations, which strongly influences tail latency, throughput, and the effectiveness of I/O parallelism.

Standardize on a single, battle-tested async runtime like Tokio to ensure a consistent concurrency model that minimizes ecosystem fragmentation and reduces cognitive load for developers. This uniformity directly enhances maintainability through reusable patterns and significantly accelerates team productivity by enabling shared expertise and streamlined debugging workflows.

/home/igor > \u250c\u2500 @1 Production Focus \u2500

\u2502 The most critical decision is choosing between single-threaded async (tokio with `current_thread`) versus multi-threaded work-stealing (tokio with `multi_thread`) based on your workload characteristics - use single-threaded for pure I/O-bound services to avoid context switching overhead, and multi-threaded for mixed I/O/CPU workloads. Equally important is designing your task spawning strategy to avoid blocking the async runtime with CPU-intensive work, using dedicated thread pools via `spawn_blocking` or `rayon` for compute-heavy operations to maintain low latency for network I/O.

\u2514\u2500

\u250c\u2500 @2 Performance Focus
\u2500

\u2502 Choosing whether to use a single shared multi-threaded async runtime (e.g., Tokio\u2019s multi-threaded scheduler) versus multiple runtimes or a custom event loop is the most important decision, because it directly affects context-switch overhead, cache locality, lock contention, and scheduler fairness under load. Closely coupled to this is deciding how to partition work between lightweight async tasks and dedicated threads for CPU-bound operations, which strongly influences tail latency, throughput, and the effectiveness of I/O parallelism.

\u2514\u2500

\u250c\u2500 @3 Maintainability Focus
\u2500

\u2502 Standardize on a single, battle-tested async runtime like Tokio to ensure a consistent concurrency model that minimizes ecosystem fragmentation and reduces cognitive load for developers. This uniformity directly enhances maintainability through reusable patterns and significantly accelerates team productivity by enabling shared expertise and streamlined debugging workflows.

\u2514\u2500

/home/igor > Sent to mshell (1832 bytes)

Received from GUI editor:

Pattern 7 Evaluator-Optimizer Loop

```rust >task
fn main() {
    println!("Write generic Rust fn binary_search<T: Ord>(slice: &[T], target: &T) \
        -> Option<usize>. Include fn main() with tests for: found, not-found, \
        empty slice, single element, duplicates. No std::slice::binary_search.");
}
```

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

<!--@1 <task >code

Write only Rust code, no fences. Must compile with rustc stable.

If there was a REJECTED verdict, fix the specific issue.

-->

```rust <code
use std::{env, fs};
fn main() {
    println!("=== Generated ===\n{}\n",
        fs::read_to_string(env::var("MSH_VAR_code").unwrap()).unwrap().trim());
}
```

<!--@2 <code >verdict

Evaluate the Rust binary search for: correct logic, proper generics with Ord,

Option<usize> return, all 5 test cases covered, no syntax errors.

If ALL pass: reply ACCEPTED

If any fail: reply REJECTED: <one sentence fix>

-->

```rust <verdict
use std::{env, fs};
fn main() {
    println!("=== Verdict: {} ===\n",
        fs::read_to_string(env::var("MSH_VAR_verdict").unwrap()).unwrap().trim());
}
```

<!--@end_loop-->

```rust <code
use std::{env, fs, process::Command};
fn main() {
    let source = fs::read_to_string(env::var("MSH_VAR_code").unwrap()).unwrap();
    println!("=== Final Accepted Code ===\n{}\n", source.trim());
    let src = "/tmp/mshell_p7_final.rs";
    let bin = "/tmp/mshell_p7_final_bin";
    fs::write(src, &source).unwrap();
    let c = Command::new("rustc").args([src, "-o", bin]).output().unwrap();
    if !c.status.success() {
        eprintln!("Compile error:\n{}", String::from_utf8_lossy(&c.stderr));
        return;
    }
    println!("=== Executing ===");
    print!("{}", String::from_utf8_lossy(&Command::new(bin).output().unwrap().stdout));
}
```

Write generic Rust fn binary_search<T: Ord>(slice: &[T], target: &T) -> Option<usize>.
Include fn main() with tests for: found, not-found, empty slice, single element, duplicates.
No std::slice::binary_search.

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

```
fn binary_search<T: Ord>(slice: &[T], target: &T) -> Option<usize> {
    if slice.is_empty() {
        return None;
    }
    let mut left = 0;
    let mut right = slice.len();
    while left < right {
        let mid = left + (right - left) / 2;
        match slice[mid].cmp(target) {
            std::cmp::Ordering::Equal => return Some(mid),
            std::cmp::Ordering::Less => left = mid + 1,
            std::cmp::Ordering::Greater => right = mid,
        }
    }
    None
}


fn main() {
    // Test found
    let arr = [1, 3, 5, 7, 9, 11];
    assert_eq!(binary_search(&arr, &5), Some(2));
    assert_eq!(binary_search(&arr, &1), Some(0));
    assert_eq!(binary_search(&arr, &11), Some(5));
    // Test not found
```

```
    assert_eq!(binary_search(&arr, &4), None);
    assert_eq!(binary_search(&arr, &0), None);
    assert_eq!(binary_search(&arr, &12), None);
    // Test empty slice
    let empty: &[i32] = &[];
    assert_eq!(binary_search(empty, &5), None);
    // Test single element
    let single = [42];
    assert_eq!(binary_search(&single, &42), Some(0));
    assert_eq!(binary_search(&single, &10), None);
    // Test duplicates
    let dups = [1, 2, 2, 2, 5, 8];
    let result = binary_search(&dups, &2);
    assert!(result.is_some());
    assert!(matches!(result, Some(1) | Some(2) | Some(3)));

    println!("All tests passed!");
}
=== Generated ===
fn binary_search<T: Ord>(slice: &[T], target: &T) -> Option<usize> {
    if slice.is_empty() {
        return None;
    }
    let mut left = 0;
    let mut right = slice.len();
    while left < right {
        let mid = left + (right - left) / 2;
```

```
        match slice[mid].cmp(target) {
            std::cmp::Ordering::Equal => return Some(mid),
            std::cmp::Ordering::Less => left = mid + 1,
            std::cmp::Ordering::Greater => right = mid,
        }
    }
    None
}
fn main() {
    // Test found
    let arr = [1, 3, 5, 7, 9, 11];
    assert_eq!(binary_search(&arr, &5), Some(2));
    assert_eq!(binary_search(&arr, &1), Some(0));
    assert_eq!(binary_search(&arr, &11), Some(5));
    // Test not found
    assert_eq!(binary_search(&arr, &4), None);
    assert_eq!(binary_search(&arr, &0), None);
    assert_eq!(binary_search(&arr, &12), None);
    // Test empty slice
    let empty: &[i32] = &[];
    assert_eq!(binary_search(empty, &5), None);
    // Test single element
    let single = [42];
    assert_eq!(binary_search(&single, &42), Some(0));
    assert_eq!(binary_search(&single, &10), None);
    // Test duplicates
    let dups = [1, 2, 2, 2, 5, 8];
```

```
    let result = binary_search(&dups, &2);
    assert!(result.is_some());
    assert!(matches!(result, Some(1) | Some(2) | Some(3)));
    println!("All tests passed!");
}
```

ACCEPTED

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

```
/home/igor > === Final Accepted Code ===
fn binary_search<T: Ord>(slice: &[T], target: &T) -> Option<usize> {
    if slice.is_empty() {
        return None;
    }
    let mut left = 0;
    let mut right = slice.len();
    while left < right {
        let mid = left + (right - left) / 2;
        match slice[mid].cmp(target) {
            std::cmp::Ordering::Equal => return Some(mid),
            std::cmp::Ordering::Less => left = mid + 1,
            std::cmp::Ordering::Greater => right = mid,
        }
    }
    None
}
fn main() {
    // Test found
```

```
    let arr = [1, 3, 5, 7, 9, 11];
    assert_eq!(binary_search(&arr, &5), Some(2));
    assert_eq!(binary_search(&arr, &1), Some(0));
    assert_eq!(binary_search(&arr, &11), Some(5));
    // Test not found
    assert_eq!(binary_search(&arr, &4), None);
    assert_eq!(binary_search(&arr, &0), None);
    assert_eq!(binary_search(&arr, &12), None);
    // Test empty slice
    let empty: &[i32] = &[];
    assert_eq!(binary_search(empty, &5), None);
    // Test single element
    let single = [42];
    assert_eq!(binary_search(&single, &42), Some(0));
    assert_eq!(binary_search(&single, &10), None);
    // Test duplicates
    let dups = [1, 2, 2, 2, 5, 8];
    let result = binary_search(&dups, &2);
    assert!(result.is_some());
    assert!(matches!(result, Some(1) | Some(2) | Some(3)));
    println!("All tests passed!");
}
```

=== Final Accepted Code ===

```
fn binary_search<T: Ord>(slice: &[T], target: &T) -> Option<usize> {
    if slice.is_empty() {
        return None;
```

```rust
    }
    let mut left = 0;
    let mut right = slice.len();
    while left < right {
        let mid = left + (right - left) / 2;
        match slice[mid].cmp(target) {
            std::cmp::Ordering::Equal => return Some(mid),
            std::cmp::Ordering::Less => left = mid + 1,
            std::cmp::Ordering::Greater => right = mid,
        }
    }
    None
}
fn main() {
    // Test found
    let arr = [1, 3, 5, 7, 9, 11];
    assert_eq!(binary_search(&arr, &5), Some(2));
    assert_eq!(binary_search(&arr, &1), Some(0));
    assert_eq!(binary_search(&arr, &11), Some(5));
    // Test not found
    assert_eq!(binary_search(&arr, &4), None);
    assert_eq!(binary_search(&arr, &0), None);
    assert_eq!(binary_search(&arr, &12), None);
    // Test empty slice
    let empty: &[i32] = &[];
    assert_eq!(binary_search(empty, &5), None);
    // Test single element
```

```
    let single = [42];
    assert_eq!(binary_search(&single, &42), Some(0));
    assert_eq!(binary_search(&single, &10), None);
    // Test duplicates
    let dups = [1, 2, 2, 2, 5, 8];
    let result = binary_search(&dups, &2);
    assert!(result.is_some());
    assert!(matches!(result, Some(1) | Some(2) | Some(3)));
    println!("All tests passed!");
}
```

=== Executing ===

All tests passed!

=== Executing ===

All tests passed!

/home/igor > Sent to mshell (2932 bytes)

Received from GUI editor:

Pattern 8 Multi-Stage Rust + Multi-Model

```rust >matrix_data
fn det3(a: [[f64;3];3]) -> f64 {
    a[0][0]*(a[1][1]*a[2][2]-a[1][2]*a[2][1])
   -a[0][1]*(a[1][0]*a[2][2]-a[1][2]*a[2][0])
   +a[0][2]*(a[1][0]*a[2][1]-a[1][1]*a[2][0])
}
fn det4(a: [[f64;4];4]) -> f64 {
```

```
    (0..4).map(|j| {
        let sign = if j%2==0 { 1.0 } else { -1.0 };
        let mut minor = [[0.0f64;3];3];
        for r in 1..4 { let mut ci=0; for c in 0..4 { if c==j { continue; }
            minor[r-1][ci]=a[r][c]; ci+=1; } }
        sign * a[0][j] * det3(minor)
    }).sum()
}
fn main() {
    let m: [[f64;4];4] = [[4.,3.,2.,1.],[3.,6.,1.,2.],[2.,1.,8.,3.],[1.,2.,3.,9.]];
    println!("determinant={:.4}", det4(m));
    println!("trace={:.1}", m[0][0]+m[1][1]+m[2][2]+m[3][3]);
    println!("values={}", (0..4).flat_map(|r| (0..4).map(move |c| m[r][c].to_string()))
        .collect::<Vec<_>>().join(","));
}
```
```rust <matrix_data >eigen_stats
use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_matrix_data").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let mut vals = vec![];
    for line in raw.lines() {
        if let Some(v) = line.strip_prefix("values=") {
            vals = v.split(',').map(|s| s.parse::<f64>().unwrap()).collect::<Vec<_>>();
        }
    }
```

```
    let m: Vec<Vec<f64>> = (0..4).map(|r| vals[r*4..r*4+4].to_vec()).collect();
    let mut v = vec![1.0f64; 4];
    let mut eigenvalue = 0.0f64;
    for _ in 0..50 {
        let mut mv = vec![0.0f64; 4];
        for i in 0..4 { for j in 0..4 { mv[i] += m[i][j] * v[j]; } }
        eigenvalue = mv.iter().cloned().fold(f64::NEG_INFINITY, f64::max);
        let norm = mv.iter().map(|x| x*x).sum::<f64>().sqrt();
        v = mv.iter().map(|x| x/norm).collect();
    }
    println!("dominant_eigenvalue={:.4}", eigenvalue);
    println!("frobenius_norm={:.4}", vals.iter().map(|x| x*x).sum::<f64>().sqrt());
}
```

<!--@1 <eigen_stats >analysis

The input contains eigenvalue and Frobenius norm statistics for a 4x4 positive definite matrix.

Explain the mathematical significance of these values in one paragraph.

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

-->

<!--@2 <analysis >summary

The input is a mathematical analysis paragraph.

Compress it into a 5-word technical tweet summary.

Reply with ONLY the 5 words \u2014 do not leave the response empty.

-->

```rust &lt;matrix_data &lt;eigen_stats &lt;analysis &lt;summary
use std::{env, fs};
```

```
fn slurp(var: &str) -> String {
    env::var(var).ok()
        .and_then(|p| fs::read_to_string(&p).ok())
        .unwrap_or_else(|| "[not available]".to_string())
}

fn main() {
    for (label, var) in &[
        ("Matrix",   "MSH_VAR_matrix_data"),
        ("Eigen",    "MSH_VAR_eigen_stats"),
        ("Analysis", "MSH_VAR_analysis"),
        ("Summary",  "MSH_VAR_summary"),
    ] {
        println!("[{}]\n{}\n", label, slurp(var).trim());
    }
}
```

----

determinant=760.0000

trace=27.0

values=4,3,2,1,3,6,1,2,2,1,8,3,1,2,3,9

dominant_eigenvalue=8.6761

frobenius_norm=15.9060

The dominant eigenvalue of 8.6761 indicates the maximum rate of expansion along the principal eigenvector direction, which is crucial for understanding the matrix's spectral properties and conditioning - since this is a positive definite matrix, all eigenvalues are positive, and this largest value determines the matrix's spectral radius and influences convergence rates in iterative algorithms. The Frobenius norm of 15.9060 measures the matrix's overall "size" as the square root of the sum of all squared entries, providing insight into the matrix's energy or magnitude; when compared to the dominant eigenvalue, this relationship suggests the matrix has moderate spread in its singular values, as the

Frobenius norm (15.9060) is not dramatically larger than \u221a4 times the dominant eigenvalue (\u224818.6), indicating the matrix doesn't have extreme condition number issues and the energy is reasonably distributed across its components rather than being concentrated in a single dominant direction.

Dominant eigenvalue, Frobenius norm imply well-conditioned

/home/igor > [Matrix]

determinant=760.0000

trace=27.0

values=4,3,2,1,3,6,1,2,2,1,8,3,1,2,3,9

[Eigen]

dominant_eigenvalue=8.6761

frobenius_norm=15.9060

[Analysis]

The dominant eigenvalue of 8.6761 indicates the maximum rate of expansion along the principal eigenvector direction, which is crucial for understanding the matrix's spectral properties and conditioning - since this is a positive definite matrix, all eigenvalues are positive, and this largest value determines the matrix's spectral radius and influences convergence rates in iterative algorithms. The Frobenius norm of 15.9060 measures the matrix's overall "size" as the square root of the sum of all squared entries, providing insight into the matrix's energy or magnitude; when compared to the dominant eigenvalue, this relationship suggests the matrix has moderate spread in its singular values, as the Frobenius norm (15.9060) is not dramatically larger than \u221a4 times the dominant eigenvalue (\u224818.6), indicating the matrix doesn't have extreme condition number issues and the energy is reasonably distributed across its components rather than being concentrated in a single dominant direction.

[Summary]

Dominant eigenvalue, Frobenius norm imply well-conditioned

/home/igor > Sent to mshell (1774 bytes)

Received from GUI editor:

Pattern 9 Routing: LLM Classifies \u2192 Conditional Rust Branch

----

```rust >input
fn main() {
    println!("What is the most efficient way to sort a large Vec<u64> \
        in Rust when data has many repeated values?");
}
```

<!--@1 <input >route

Classify this Rust question into exactly one word: ALGO, MEMORY, or CONCURRENCY.

Reply with only the single classification word, nothing else.

You MUST respond with exactly one word \u2014 do not leave the response empty.

-->

```rust &lt;route if=route:ALGO
use std::time::Instant;
fn main() {
    let mut data: Vec<u64> = (0..100_000).map(|i| i % 1000).collect();
    let t0 = Instant::now();
    data.sort_unstable();
    println!("=== ALGO: Sort Benchmark ===");
    println!("Sorted 100k elements in {:?}", t0.elapsed());
    println!("pdqsort optimal for low-cardinality data");
}
```

```rust <route if=route:MEMORY
fn main() {
    println!("=== MEMORY: Allocation Decision Tree ===");
    println!("Box<T>  \u{2192} single owner, zero overhead");
    println!("Rc<T>   \u{2192} shared, single-thread");
```

~~~~
    println!("Arc<T>  \u{2192} shared, multi-thread safe");
}
```
```rust <route if=route:CONCURRENCY
use std::{time::Instant, thread};
fn main() {
    let t0 = Instant::now();
    let h: Vec<_> = (0..100).map(|i| thread::spawn(move || i*2)).collect();
    for h in h { h.join().unwrap(); }
    println!("=== CONCURRENCY: Thread Benchmark ===");
    println!("100 OS threads spawn+join: {:?}", t0.elapsed());
}
```
```rust <route
use std::{env, fs};
fn main() {
    let route = env::var("MSH_VAR_route").ok()
        .and_then(|p| fs::read_to_string(&p).ok())
        .unwrap_or_else(|| "[not classified]".to_string());
    println!("Classified as: {}", route.trim());
}
```
----------------------------------------
What is the most efficient way to sort a large Vec<u64> in Rust when data has many
repeated values?
ALGO
/home/igor > Classified as: ALGO
~~~~

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

Received from GUI editor:

Pattern 10 Full Pipeline: All Patterns Combined

````
```rust >raw_data
fn is_prime(n: u64) -> bool {
    if n < 2 { return false; }
    if n == 2 || n == 3 { return true; }
    if n%2==0 || n%3==0 { return false; }
    let mut i = 5u64;
    while i*i <= n { if n%i==0 || n%(i+2)==0 { return false; } i+=6; }
    true
}
fn main() {
    let mersennes: Vec<u64> = (2u32..=19)
        .filter(|&p| is_prime(p as u64))
        .map(|p| (1u64 << p) - 1)
        .filter(|&m| is_prime(m))
        .collect();
    println!("{}", mersennes.iter().map(|n| n.to_string()).collect::<Vec<_>>().join(","));
}
```
```rust <raw_data >stats
use std::{env, fs};
fn main() {
````
````

````
    let path = env::var("MSH_VAR_raw_data").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let nums: Vec<u64> = raw.trim().split(',').map(|s| s.trim().parse().unwrap()).collect();
    let sum: u64 = nums.iter().sum();
    println!("count={} sum={} min={} max={}", nums.len(), sum,
        nums.iter().min().unwrap(), nums.iter().max().unwrap());
    let forms: Vec<String> = nums.iter().map(|&n| {
        let p = (n+1).next_power_of_two().trailing_zeros();
        format!("2^{}-1", p)
    }).collect();
    println!("forms={}", forms.join(", "));
}
```
<!--@1 <stats >analysis
The input contains statistics about Mersenne prime numbers.
In one sentence, describe what makes Mersenne primes mathematically special.
-->
<!--@2 <raw_data >poem
Write a 2-line rhyming poem about the Mersenne prime numbers in the input.
-->
```rust <analysis <poem
use std::{env, fs};
fn main() {
    for (l, v) in &[("Analysis","analysis"),("Poem","poem")] {
        println!("=== {} ===\n{}\n", l,
            fs::read_to_string(env::var(&format!("MSH_VAR_{}", v)).unwrap()).unwrap().trim());
    }
````

```
}
```

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

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

captures both the mathematical and aesthetic beauty of Mersenne primes.

-->

```rust <combined
use std::{env, fs};

fn main() {

    let text = fs::read_to_string(env::var("MSH_VAR_combined").unwrap()).unwrap();

    let text = text.trim();

    let w = text.len().max(50) + 4;

    println!("\u{2554}{}\u{2557}", "\u{2550}".repeat(w-2));

    println!("\u{2551}{:^width$}\u{2551}", " Mersenne Primes \u{2014} Final Synthesis ", width=w-2);

    println!("\u{2560}{}\u{2563}", "\u{2550}".repeat(w-2));

    println!("\u{2551}  {:<width$}  \u{2551}", text, width=w-6);

    println!("\u{255a}{}\u{255d}", "\u{2550}".repeat(w-2));

}
```

3,7,31,127,8191,131071,524287

count=7 sum=663717 min=3 max=524287

forms=2^2-1, 2^3-1, 2^5-1, 2^7-1, 2^13-1, 2^17-1, 2^19-1

Mersenne primes are mathematically special because they are prime numbers of the form 2^p - 1 where p itself must be prime, and they are intrinsically connected to perfect numbers since every even perfect number corresponds to a Mersenne prime via Euclid's formula 2^(p-1)(2^p - 1).

From 3, 7, 31 they climb, strange stars in binary\u2019s gleaming clime,

To 127 through 524287 they chime, rare Mersenne whispers that perfectly rhyme.

=== Analysis ===

Mersenne primes are mathematically special because they are prime numbers of the form 2^p - 1 where p itself must be prime, and they are intrinsically connected to perfect numbers since every even perfect number corresponds to a Mersenne prime via Euclid's formula 2^(p-1)(2^p - 1).

=== Poem ===

From 3, 7, 31 they climb, strange stars in binary\u2019s gleaming clime,

To 127 through 524287 they chime, rare Mersenne whispers that perfectly rhyme.

Mersenne primes are mathematically special because they are prime numbers of the form 2^p - 1 where p itself must be prime, creating an elegant sequence\u20143, 7, 31, 127, 8191, 131071, 524287\u2014that ascends like rare binary stars through mathematical space, each perfectly rhyming with Euclid's timeless formula 2^(p-1)(2^p - 1) to generate every even perfect number.

\u2554\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255

0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2557

\u2551
Mersenne Primes \u2014 Final Synthesis
\u2551

\u2560\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2563

\u2551 Mersenne primes are mathematically special because they are prime numbers of the form 2^p - 1 where p itself must be prime, creating an elegant sequence\u20143, 7, 31, 127, 8191, 131071, 524287\u2014that ascends like rare binary stars through mathematical space, each perfectly rhyming with Euclid's timeless formula 2^(p-1)(2^p - 1) to generate every even perfect number.   \u2551

\u255a\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\
u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550
\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u255d

/home/igor > Sent to mshell (860 bytes)

Received from GUI editor:

Pattern 11 MShell Node with Multiple Models

````
```rust >topic
fn main() { println!("Rust ownership and borrowing system"); }
```

```rust >style
fn main() { println!("precise and example-driven for systems programmers"); }
```

```mshell <topic <style >explanation
ollama1 "Explain '$topic' in a '$style' way in two sentences"
```

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

```rust <explanation <keywords
use std::{env, fs};

fn main() {
    let exp = fs::read_to_string(env::var("MSH_VAR_explanation").unwrap()).unwrap();
    let kw  = fs::read_to_string(env::var("MSH_VAR_keywords").unwrap()).unwrap();
    println!("=== Explanation ===\n{}\n", exp.trim());
    println!("=== Keywords ===\n{}", kw.trim());
}
```
````

Rust ownership and borrowing system

precise and example-driven for systems programmers

Rust's ownership system ensures memory safety by enforcing that each value has exactly one owner at compile time, automatically deallocating when the owner goes out of scope, while borrowing allows temporary access through references (&T for immutable, &mut T for mutable) that must follow the rule: either multiple immutable references OR one mutable reference, never both simultaneously. For example, `let mut vec = Vec::new(); let r1 = &vec; let r2 = &vec;` compiles fine, but adding `let r3 = &mut vec;` while r1/r2 are still in scope fails compilation, preventing data races and use-after-free bugs that plague C/C++ without runtime overhead.ownership,borrowing,immutability,mutability/home/igor > === Explanation ===

Rust's ownership system ensures memory safety by enforcing that each value has exactly one owner at compile time, automatically deallocating when the owner goes out of scope, while borrowing allows temporary access through references (&T for immutable, &mut T for mutable) that must follow the rule: either multiple immutable references OR one mutable reference, never both simultaneously. For example, `let mut vec = Vec::new(); let r1 = &vec; let r2 = &vec;` compiles fine, but adding `let r3 = &mut vec;` while r1/r2 are still in scope fails compilation, preventing data races and use-after-free bugs that plague C/C++ without runtime overhead.

=== Keywords ===

ownership,borrowing,immutability,mutability

/home/igor > Sent to mshell (3398 bytes)

Received from GUI editor:

Pattern 12 Async Parallel 3 Models + Await Barrier + Synthesis

----

```rust >question
fn main() {
    println!("What is the difference between dyn Trait and <T: Trait> in Rust, \
        and when should you use each? Answer in one sentence.");
}
```

<!--@1 <question >ans1 async

Explain in one sentence using an intuitive analogy for a Rust beginner.

You MUST respond with exactly one sentence \u2014 do not leave the response empty.

-->

<!--@2 <question >ans2 async

Explain in one sentence using a memory layout perspective.

You MUST respond with exactly one sentence \u2014 do not leave the response empty.

-->

<!--@3 <question >ans3 async

Explain in one sentence using compiler type-theory terminology.

You MUST respond with exactly one sentence \u2014 do not leave the response empty.

-->

```bash await=ans1,ans2,ans3

for p in "$(printenv MSH_VAR_ans1)" "$(printenv MSH_VAR_ans2)" "$(printenv MSH_VAR_ans3)"; do

  i=0

  while [ ! -s "$p" ] && [ $i -lt 180 ]; do sleep 1; i=$((i+1)); done

  if [ ! -s "$p" ]; then echo "[no response]" > "$p"; fi

done

```

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

Synthesize the three explanations into one perfect sentence.

You MUST respond with exactly one sentence \u2014 do not leave the response empty.

-->

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

pf=$(printenv MSH_VAR_final)

i=0

while [ ! -s "$pf" ] && [ $i -lt 180 ]; do sleep 1; i=$((i+1)); done
```

```
if [ ! -s "$pf" ]; then echo "[synthesis not available]" > "$pf"; fi

echo "\u250c\u2500 Beginner Intuition (@1)"; cat $(printenv MSH_VAR_ans1); echo
"\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500"

echo "\u250c\u2500 Memory Layout (@2)";      cat $(printenv MSH_VAR_ans2); echo
"\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500"

echo "\u250c\u2500 Type Theory (@3)";       cat $(printenv MSH_VAR_ans3); echo
"\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500"

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

cat $(printenv MSH_VAR_final)

echo
"\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u255
0\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u25
50\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2
550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550\u
2550\u2550\u2550\u2550\u2550\u2550\u2550\u2550"
```

[async llm] Launched PID 17752 \u2192 var=ans1 (model @1)

[async llm] Launched PID 17753 \u2192 var=ans2 (model @2)

[async llm] Launched PID 17755 \u2192 var=ans3 (model @3)

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

[async] Waiting for PID 17752 (var=ans1)...

[async] PID 17752 done (var=ans1)

Choosing the right async runtime is like deciding between having one super-efficient waiter (single-threaded) who never gets distracted but can only serve one table at a time, versus having a team of waiters (multi-threaded) who can work on different tables simultaneously but need to coordinate with each other\u2014pick single-threaded for pure I/O tasks and multi-threaded when you have mixed workloads.

The most critical architectural decision for high-throughput network services is standardizing on a single, battle-tested async runtime like Tokio while choosing the appropriate concurrency model\u2014single-threaded for pure I/O efficiency versus multi-threaded work-stealing for mixed workloads\u2014because this determines task-to-core mapping, memory layout, and cache locality patterns that directly impact performance at scale, while ensuring ecosystem consistency and team productivity through shared expertise and streamlined development workflows.

\u250c\u2500 Beginner Intuition (@1)

Choosing the right async runtime is like deciding between having one super-efficient waiter (single-threaded) who never gets distracted but can only serve one table at a time, versus having a team of waiters (multi-threaded) who can work on different tables simultaneously but need to coordinate with each other\u2014pick single-threaded for pure I/O tasks and multi-threaded when you have mixed workloads.

\u2514\u2500

\u250c\u2500 Memory Layout (@2)

Choosing between a single shared multi-threaded async runtime and a design that shards work across multiple runtimes or dedicated worker threads is critical because it determines how tasks are mapped to cores, how stacks, heaps, and task state are laid out and reused in memory, and thus how much cache locality, allocation overhead, and contention you incur at scale.\u2514\u2500

\u250c\u2500 Type Theory (@3)

Standardize on a single, battle-tested async runtime like Tokio to ensure a consistent concurrency model that minimizes ecosystem fragmentation and reduces cognitive load for developers. This uniformity directly enhances maintainability through reusable patterns and significantly accelerates team productivity by enabling shared expertise and streamlined debugging workflows.\u2514\u2500

\u2550

The most critical architectural decision for high-throughput network services is standardizing on a single, battle-tested async runtime like Tokio while choosing the appropriate concurrency model\u2014single-threaded for pure I/O efficiency versus multi-threaded work-stealing for mixed workloads\u2014because this determines task-to-core mapping, memory layout, and cache locality patterns that directly impact performance at scale, while ensuring ecosystem consistency and team productivity through shared expertise and streamlined development workflows.\u2550

/home/igor > Sent to mshell (1393 bytes)

Received from GUI editor:

Pattern 13 WHILE Loop: Iterative Counter with LLM Commentary

```rust >status

fn main() { println!("running"); }

```

```rust >counter
```

````
fn main() { println!("0"); }
```
<!--@while status:running-->
```rust <counter <status >counter >status
use std::{env, fs};
fn main() {
    let counter_path = env::var("MSH_VAR_counter").unwrap();
    let status_path  = env::var("MSH_VAR_status").unwrap();
    let raw = fs::read_to_string(&counter_path).unwrap_or_else(|_| "0".into());
    let val: u64 = raw.trim().parse().unwrap_or(0);
    let new_val = val + 1;
    fs::write(&counter_path, format!("{}", new_val)).unwrap();
    fs::write(&status_path, if new_val >= 4 { "done" } else { "running" }).unwrap();
    print!("{}", new_val);
}
```
<!--@1 <counter >comment
The input is a single integer. State one interesting mathematical fact about
this specific number. One sentence only.
-->
```rust <comment
use std::{env, fs};
fn main() {
    let counter = fs::read_to_string(env::var("MSH_VAR_counter").unwrap()).unwrap();
    let comment = fs::read_to_string(env::var("MSH_VAR_comment").unwrap()).unwrap();
    println!("[iter {}] {}", counter.trim(), comment.trim());
}
````

```

&lt;!--@end_while--&gt;

```rust &lt;counter

use std::{env, fs};

fn main() {

    println!("=== WHILE complete. Final counter = {} ===",

        fs::read_to_string(env::var("MSH_VAR_counter").unwrap()).unwrap().trim());

}

```

----

running

0

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

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

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

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

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

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

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

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

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

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

4The number 4 is the smallest composite number and the only number that equals both 2^2 and 2+2, making it unique as a perfect square that can also be expressed as the sum of two identical primes.

[iter 4] The number 4 is the smallest composite number and the only number that equals both 2^2 and 2+2, making it unique as a perfect square that can also be expressed as the sum of two identical primes.

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

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

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

Received from GUI editor:

Pattern 14 FOREACH: LLM Processes Each Item in a List

```rust >primitives
fn main() {
    print!("Mutex\nRwLock\nAtomicUsize\nMpscChannel\nBarrier");
}
```

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

<!--@1 <item >description

The input is a Rust concurrency primitive name.

In one sentence, describe what it does, its key guarantee, and one ideal use case.

-->

```rust <description
use std::{env, fs};
fn main() {
    let item = fs::read_to_string(env::var("MSH_VAR_item").unwrap()).unwrap();
```

let desc = fs::read_to_string(env::var("MSH_VAR_description").unwrap()).unwrap();

let name = item.trim();

println!("\u{250c}\u{2500} {} {}", name,
"\u{2500}".repeat(50usize.saturating_sub(name.len()+3)));

println!("\u{2502} {}", desc.trim());

println!("\u{2514}{}\n", "\u{2500}".repeat(52));

}

```

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

```rust

fn main() { println!("=== All 5 Rust concurrency primitives described ==="); }

```

Mutex

RwLock

AtomicUsize

MpscChannel

Barrier[foreach] iter 1: item=Mutex

Mutex provides mutually exclusive access to shared data by allowing only one thread to hold the lock at a time, guaranteeing memory safety and preventing data races, making it ideal for protecting shared counters or configuration state that multiple threads need to read and modify.

\u250c\u2500 Mutex
\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500

\u2502 Mutex provides mutually exclusive access to shared data by allowing only one thread to hold the lock at a time, guaranteeing memory safety and preventing data races, making it ideal for protecting shared counters or configuration state that multiple threads need to read and modify.

\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
```

00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500

[foreach] iter 2: item=RwLock

RwLock allows multiple concurrent readers or a single exclusive writer to access shared data, guaranteeing that reads never conflict with each other but are mutually exclusive with writes, making it ideal for caching scenarios where data is read frequently but updated infrequently.

\u250c\u2500 RwLock
\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500

\u2502 RwLock allows multiple concurrent readers or a single exclusive writer to access shared data, guaranteeing that reads never conflict with each other but are mutually exclusive with writes, making it ideal for caching scenarios where data is read frequently but updated infrequently.

\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500

[foreach] iter 3: item=AtomicUsize

AtomicUsize provides lock-free atomic operations on a usize value, guaranteeing that reads, writes, and modifications are indivisible across threads without requiring mutex overhead, making it ideal for implementing high-performance counters, reference counts, or simple shared state that needs frequent concurrent updates.

\u250c\u2500 AtomicUsize
\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500

\u2502 AtomicUsize provides lock-free atomic operations on a usize value, guaranteeing that reads, writes, and modifications are indivisible across threads without requiring mutex overhead, making it ideal for implementing high-performance counters, reference counts, or simple shared state that needs frequent concurrent updates.

\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25

00\u2500

[foreach] iter 4: item=MpscChannel

MpscChannel (Multiple Producer, Single Consumer) enables multiple threads to send messages to a single receiving thread through a thread-safe queue, guaranteeing ordered delivery and memory safety without explicit locking, making it ideal for worker thread patterns where multiple tasks need to send results or work items to a central coordinator thread.

\u250c\u2500 MpscChannel \u2500

\u2502 MpscChannel (Multiple Producer, Single Consumer) enables multiple threads to send messages to a single receiving thread through a thread-safe queue, guaranteeing ordered delivery and memory safety without explicit locking, making it ideal for worker thread patterns where multiple tasks need to send results or work items to a central coordinator thread.

\u2514\u2500

[foreach] iter 5: item=Barrier

Barrier synchronizes multiple threads by blocking them until all threads reach the same synchronization point, guaranteeing that no thread proceeds past the barrier until a specified number of threads have all arrived, making it ideal for parallel algorithms where all threads must complete a phase of work before any can begin the next phase.

\u250c\u2500 Barrier \u2500

\u2502 Barrier synchronizes multiple threads by blocking them until all threads reach the same synchronization point, guaranteeing that no thread proceeds past the barrier until a specified number of threads have all arrived, making it ideal for parallel algorithms where all threads must complete a phase of work before any can begin the next phase.

\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500

/home/igor > === All 5 Rust concurrency primitives described ===

/home/igor > Sent to mshell (1845 bytes)

Received from GUI editor:

Pattern 15 TRY/CATCH: Safe Execution with Error Capture

```rust >input
fn main() {
    println!(r#"{{name: "Ferris", lang: "Rust", version: 1.75"#);
}
```

<!--@try-->

```rust <input
use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_input").unwrap();
    let s = fs::read_to_string(&path).unwrap();
    let s = s.trim();
    if !s.starts_with('{') || !s.ends_with('}') {
        eprintln!("JSON error: not a valid object");
        std::process::exit(1);
    }
```

````
    for part in s.trim_matches(|c| c == '{' || c == '}').split(',') {
        let kv: Vec<&str> = part.splitn(2, ':').collect();
        if kv.len() != 2 { eprintln!("JSON error: bad kv"); std::process::exit(1); }
        let key = kv[0].trim();
        if !key.starts_with('"') { eprintln!("JSON error: unquoted key: {}", key);
std::process::exit(1); }
    }
    println!("Parsed OK");
}
```
<!--@catch >error-->
```rust
fn main() {
    println!("=== Caught: try_block_failed ===");
    println!("JSON parse failed. Activating safe fallback.");
}
```
<!--@end_try-->
```rust <input >safe_result
use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_input").unwrap();
    let raw = fs::read_to_string(&path).unwrap();
    let cleaned = raw.trim().trim_matches(|c| c=='{' || c=='}').replace('"', "");
    let fields: Vec<String> = cleaned.split(',').filter_map(|p| {
        let kv: Vec<&str> = p.splitn(2, ':').collect();
        if kv.len()==2 { Some(format!("{}={}", kv[0].trim(), kv[1].trim())) } else { None }
````

```
    }).collect();

    println!("Safe fallback: {} fields: {}", fields.len(), fields.join(" | "));

}
```

```rust &lt;safe_result
use std::{env, fs};

fn main() {

    println!("=== Safe Result ===\n{}",

        fs::read_to_string(env::var("MSH_VAR_safe_result").unwrap()).unwrap().trim());

}
```

{name: "Ferris", lang: "Rust", version: 1.75

[try] executing try block

JSON error: not a valid object

[try] try block failed, executing catch block

=== Caught: try_block_failed ===

JSON parse failed. Activating safe fallback.

Safe fallback: 3 fields: name=Ferris | lang=Rust | version=1.75

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

Safe fallback: 3 fields: name=Ferris | lang=Rust | version=1.75

/home/igor > Sent to mshell (2051 bytes)

Received from GUI editor:

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

```rust >dataset
fn main() {
   print!("12,15,11,18,14,13,16,10,19,12\n45,38,52,41,47,50,43,39,55,44");
}
```

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

<!--@1 <dataset_1 >analysis1 async

The input contains comma-separated benchmark latency measurements in nanoseconds.

Analyze the distribution: mean, outliers, stability. One paragraph.

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

-->

<!--@2 <dataset_2 >analysis2 async

The input contains comma-separated benchmark latency measurements in nanoseconds.

Analyze the distribution: mean, outliers, stability. One paragraph.

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

-->

```bash await=analysis1,analysis2
for p in "$(printenv MSH_VAR_analysis1)" "$(printenv MSH_VAR_analysis2)"; do
   i=0; while [ ! -s "$p" ] && [ $i -lt 180 ]; do sleep 1; i=$((i+1)); done
   if [ ! -s "$p" ]; then echo "[no response]" > "$p"; fi
done
```

<!--@merge-->

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

Compare two benchmark suite analyses: which is faster, which is more stable,

what does this suggest? Two sentences.

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

-->

```rust <combined
use std::{env, fs};

fn stats(data: &str) -> (f64, f64) {

    let nums: Vec<f64> = data.trim().split(',').map(|s| s.trim().parse().unwrap_or(0.0)).collect();

    let mean = nums.iter().sum::<f64>() / nums.len() as f64;

    let std_dev = (nums.iter().map(|x| (x-mean).powi(2)).sum::<f64>() / nums.len() as f64).sqrt();

    (mean, std_dev)

}

fn main() {

    for (label, var) in &[("Suite A (Parser)","dataset_1"),("Suite B (Serializer)","dataset_2")] {

        let raw = fs::read_to_string(env::var(&format!("MSH_VAR_{}", var)).unwrap()).unwrap();

        let (mean, sd) = stats(&raw);

        println!("{}: mean={:.1}ns  \u{03c3}={:.1}ns", label, mean, sd);

    }

    let combined = fs::read_to_string(env::var("MSH_VAR_combined").unwrap()).unwrap();

    println!("\n=== Synthesized ===\n{}", combined.trim());

}
```

12,15,11,18,14,13,16,10,19,12

45,38,52,41,47,50,43,39,55,44[split] dataset_1 = 12,15,11,18,14,13,16,10,19,12

[split] dataset_2 = 45,38,52,41,47,50,43,39,55,44

[async llm] Launched PID 18581 \u2192 var=analysis1 (model @1)

[async llm] Launched PID 18582 \u2192 var=analysis2 (model @2)

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

[async] Waiting for PID 18581 (var=analysis1)...

[async] PID 18581 done (var=analysis1)

The latency measurements show a mean of 14.0 nanoseconds with values ranging from 10ns to 19ns, indicating relatively stable performance with moderate variability. The distribution appears fairly normal with most values clustering around 12-16ns, though 10ns stands out as a potential low outlier (28% below mean) and 19ns as a high outlier (36% above mean). With a standard deviation of approximately 3.0ns and a coefficient of variation around 21%, this represents reasonably consistent performance for nanosecond-level operations, though the 9ns range suggests some underlying variability in the system or measurement process that could be investigated for optimization.

Analysis1 shows faster performance with a mean of 14.0ns compared to Analysis2's 45.4ns mean, making it approximately 3x faster, while Analysis2 demonstrates superior stability with no statistical outliers and a tighter relative spread (38-55ns range) versus Analysis1's wider variability and identified outliers. This suggests Analysis1 likely represents a more optimized or lower-level operation that sacrifices some consistency for raw speed, while Analysis2 represents a more predictable system that prioritizes stable performance over peak speed.

/home/igor > Suite A (Parser): mean=14.0ns \u03c3=2.8ns

Suite B (Serializer): mean=45.4ns \u03c3=5.3ns

=== Synthesized ===

Analysis1 shows faster performance with a mean of 14.0ns compared to Analysis2's 45.4ns mean, making it approximately 3x faster, while Analysis2 demonstrates superior stability with no statistical outliers and a tighter relative spread (38-55ns range) versus Analysis1's wider variability and identified outliers. This suggests Analysis1 likely represents a more optimized or lower-level operation that sacrifices some consistency for raw speed, while Analysis2 represents a more predictable system that prioritizes stable performance over peak speed.

/home/igor > Sent to mshell (1421 bytes)

Received from GUI editor:

Pattern 17 CONFIG Node: Parameterized Pipeline

```config
```

````
topic=Rust lifetimes and borrow checker
style=precise and example-driven
max_words=60
```
```rust >topic
fn main() { println!("Rust lifetimes and borrow checker"); }
```
```rust >style
fn main() { println!("precise and example-driven"); }
```
<!--@1 <topic <style >explanation
The first input is a Rust topic. The second input is a writing style.
Explain the topic in that style. Maximum 60 words.
-->
<!--@2 <explanation >keywords
Extract exactly 5 keywords as a comma-separated list, nothing else.
-->
```rust <explanation <keywords >report
use std::{env, fs};
fn main() {
    let topic   = fs::read_to_string(env::var("MSH_VAR_topic").unwrap()).unwrap();
    let style   = fs::read_to_string(env::var("MSH_VAR_style").unwrap()).unwrap();
    let exp     = fs::read_to_string(env::var("MSH_VAR_explanation").unwrap()).unwrap();
    let kw      = fs::read_to_string(env::var("MSH_VAR_keywords").unwrap()).unwrap();
    let sep = "\u{2550}".repeat(50);
    println!("{}\n{:^50}\n{}", sep, "Rust Pipeline Report", sep);
    println!("Topic   : {}", topic.trim());
````

```
    println!("Style   : {}", style.trim());

    println!("{}", "\u{2500}".repeat(50));

    println!("Explanation:\n{}", exp.trim());

    println!("{}", "\u{2500}".repeat(50));

    println!("Keywords: {}", kw.trim());

    println!("{}", sep);

}
```

[config] topic=Rust lifetimes and borrow checker

[config] style=precise and example-driven

[config] max_words=60

Rust lifetimes and borrow checker

precise and example-driven

Rust lifetimes ensure references don't outlive their data. The borrow checker enforces: one mutable reference OR multiple immutable references at any time.

```rust
let mut s = String::from("hello");

let r1 = &s;       // OK: immutable borrow

let r2 = &s;       // OK: multiple immutable

// let r3 = &mut s; // ERROR: can't mix mutable/immutable
```

Rust,lifetimes,borrow,mutable,immutable

\u2550

\u2550

Topic : Rust lifetimes and borrow checker

Style : precise and example-driven

\u2500

Explanation:

Rust lifetimes ensure references don't outlive their data. The borrow checker enforces: one mutable reference OR multiple immutable references at any time.

```rust
let mut s = String::from("hello");
let r1 = &s;       // OK: immutable borrow
let r2 = &s;       // OK: multiple immutable
// let r3 = &mut s; // ERROR: can't mix mutable/immutable
```

\u2500

Keywords: Rust,lifetimes,borrow,mutable,immutable

\u2550

/home/igor > Sent to mshell (2358 bytes)

Received from GUI editor:

Pattern 18 FOREACH + Async LLM: Parallel Batch Processing

```rust >patterns
fn main() {
    print!("Builder\nNewtype\nTypeState\nRaiiGuard");
}
```

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

<!--@1 <pattern >explanation async

The input is a Rust design pattern name.

Explain it in one sentence for a developer new to Rust.

You MUST respond with exactly one sentence \u2014 do not leave the response empty.

-->

<!--@2 <pattern >analogy async

The input is a Rust design pattern name.

Give one real-world non-programming analogy. One sentence only.

You MUST respond with exactly one sentence \u2014 do not leave the response empty.

-->

```bash await=explanation,analogy
for p in "$(printenv MSH_VAR_explanation)" "$(printenv MSH_VAR_analogy)"; do
    i=0
    while [ ! -s "$p" ] && [ $i -lt 180 ]; do sleep 1; i=$((i+1)); done
    if [ ! -s "$p" ]; then echo "[no response]" > "$p"; fi
done
```

```bash <pattern <explanation <analogy

echo
"\u250c\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2510"

printf "\u2502 Pattern: %-44s \u2502\n" "$(cat $(printenv MSH_VAR_pattern))"

echo
"\u251c\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2524"

printf "\u2502 \U0001f4a1 %-50s \u2502\n" "$(cat $(printenv MSH_VAR_explanation))"

printf "\u2502 \U0001f517 %-50s \u2502\n" "$(cat $(printenv MSH_VAR_analogy))"

echo
"\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2518"

echo ""

```

<!--@end_foreach-->

```bash

echo "=== Batch complete: 4 Rust design patterns described ==="

```

Builder

Newtype

TypeState

RaiiGuard[foreach] iter 1: pattern=Builder

[async llm] Launched PID 18782 \u2192 var=explanation (model @1)

[async llm] Launched PID 18783 \u2192 var=analogy (model @2)

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

[async] Waiting for PID 18782 (var=explanation)...

[async] PID 18782 done (var=explanation)

The Builder pattern in Rust uses method chaining to construct complex objects step-by-step, typically with each method taking `self` and returning `Self` to enable fluent APIs like `Config::new().timeout(30).retries(3).build()`, making it especially useful for structs with many optional fields.

\u250c\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500
\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2510

\u2502 Pattern: Builder \u2502

\u251c\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500
\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2524

\u2502 \U0001f4a1 The Builder pattern in Rust uses method chaining to construct complex objects step-by-step, typically with each method taking `self` and returning `Self` to enable fluent APIs like `Config::new().timeout(30).retries(3).build()`, making it especially useful for structs with many optional fields. \u2502

\u2502 \U0001f517 A builder pattern is like ordering a custom sandwich at a deli, where you specify each ingredient step by step before the chef assembles the final meal. \u2502

\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2518

[foreach] iter 2: pattern=Newtype

[async llm] Launched PID 18825 \u2192 var=explanation (model @1)

[async llm] Launched PID 18826 \u2192 var=analogy (model @2)

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

[async] Waiting for PID 18825 (var=explanation)...

[async] PID 18825 done (var=explanation)

The newtype pattern wraps an existing type in a new struct with one field to create a distinct type with additional type safety, methods, or trait implementations while maintaining zero runtime cost.

\u250c\u2500\u2510

\u2502 Pattern: Newtype \u2502

\u251c\u2500\u2524

\u2502 \U0001f4a1 The newtype pattern wraps an existing type in a new struct with one field to create a distinct type with additional type safety, methods, or trait implementations while maintaining zero runtime cost. \u2502

\u2502 \U0001f517 A Rust newtype is like putting an official stamp on an otherwise ordinary document to give it a specific legal role without changing the paper itself. \u2502

\u2514\u2500\u2518

[foreach] iter 3: pattern=TypeState

[async llm] Launched PID 18843 \u2192 var=explanation (model @1)

[async llm] Launched PID 18844 \u2192 var=analogy (model @2)

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

[async] Waiting for PID 18843 (var=explanation)...

[async] PID 18843 done (var=explanation)

TypeState is a Rust design pattern that uses the type system to enforce state transitions at compile time by encoding different states as distinct types, preventing invalid operations like trying to read from a closed file or send data through a disconnected socket.

\u250c\u2500\u2510

\u2502 Pattern: TypeState \u2502

\u251c\u2500\u2524

\u2502 \U0001f4a1 TypeState is a Rust design pattern that uses the type system to enforce state transitions at compile time by encoding different states as distinct types, preventing invalid operations like trying to read from a closed file or send data through a disconnected socket. \u2502

\u2502 \U0001f517 TypeState in Rust is like having different security badges for a building where each badge only opens doors appropriate to your current role or clearance level, preventing you from entering areas before you\u2019re properly authorized. \u2502

\u2514\u2500\u2518

[foreach] iter 4: pattern=RaiiGuard

[async llm] Launched PID 18861 \u2192 var=explanation (model @1)

[async llm] Launched PID 18862 \u2192 var=analogy (model @2)

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

[async] Waiting for PID 18861 (var=explanation)...

[async] PID 18861 done (var=explanation)

RaiiGuard is a Rust design pattern where a struct automatically cleans up resources (like releasing locks, closing files, or freeing memory) when it goes out of scope using the Drop trait, ensuring resource cleanup happens even if errors occur or early returns are taken.

\u250c\u2500\u25

00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2510

\u2502 Pattern: RaiiGuard \u2502

\u251c\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500
\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2524

\u2502 \U0001f4a1 RaiiGuard is a Rust design pattern where a struct automatically cleans up resources (like releasing locks, closing files, or freeing memory) when it goes out of scope using the Drop trait, ensuring resource cleanup happens even if errors occur or early returns are taken. \u2502

\u2502 \U0001f517 Like checking into a hotel where your room key automatically grants access when you arrive and is automatically deactivated when you check out, without you ever having to think about managing it yourself. \u2502

\u2514\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u250
0\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u25
00\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2
500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u
2500\u2500\u2500\u2500\u2500\u2500\u2500\u2500\u2518

=== Batch complete: 4 Rust design patterns described ===

/home/igor > Sent to mshell (2081 bytes)

Received from GUI editor:

Pattern 19 WHILE Quality Gate: Generate Until Threshold

```rust >task
fn main() {
   println!("Write a Rust iterator chain: take Vec<i32>, filter evens, \
       square them, collect into Vec<i32>. One expression, no fn main().");
}
```

````
```rust >status
fn main() { println!("running"); }
```
```rust >iteration
fn main() { println!("0"); }
```
```rust >score
fn main() { println!("0"); }
```
```rust >snippet
fn main() { println!(""); }
```
<!--@while status:running-->
```rust <iteration >iteration
use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_iteration").unwrap();
    let val: u64 = fs::read_to_string(&path).unwrap_or_else(|_| "0".into())
        .trim().parse().unwrap_or(0);
    let new_val = val + 1;
    fs::write(&path, format!("{}", new_val)).unwrap();
    print!("{}", new_val);
}
```
<!--@1 <task >snippet
Write ONLY the Rust expression/code. No fn main(), no fences, no explanation.
-->
````

<!--@2 <snippet >score

Rate this Rust snippet 1-10: correctness, idiomatic style, readability.

Reply with ONLY the integer, nothing else.

-->

```rust <iteration <score <snippet >status
use std::{env, fs};
fn main() {
    let iter    = fs::read_to_string(env::var("MSH_VAR_iteration").unwrap()).unwrap();
    let score   = fs::read_to_string(env::var("MSH_VAR_score").unwrap()).unwrap();
    let snippet = fs::read_to_string(env::var("MSH_VAR_snippet").unwrap()).unwrap();
    let status_path = env::var("MSH_VAR_status").unwrap();
    let score_n: u64 = score.trim().parse().unwrap_or(0);
    println!("[Iter {}] Score={}/10  Snippet: {}", iter.trim(), score_n, snippet.trim());
    fs::write(&status_path, if score_n >= 8 { "done" } else { "running" }).unwrap();
}
```

<!--@end_while-->

```rust <snippet <score
use std::{env, fs};
fn main() {
    let snippet = fs::read_to_string(env::var("MSH_VAR_snippet").unwrap()).unwrap();
    let score   = fs::read_to_string(env::var("MSH_VAR_score").unwrap()).unwrap();
    println!("=== Accepted (score={}/10) ===\n{}", score.trim(), snippet.trim());
}
```

Write a Rust iterator chain: take Vec<i32>, filter evens, square them, collect into Vec<i32>. One expression, no fn main().

running

0

0

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

1vec.into_iter().filter(|&x| x % 2 == 0).map(|x| x * x).collect::<Vec<i32>>()

9

[Iter 1] Score=9/10 Snippet: vec.into_iter().filter(|&x| x % 2 == 0).map(|x| x * x).collect::<Vec<i32>>()

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

/home/igor > === Accepted (score=9/10) ===

vec.into_iter().filter(|&x| x % 2 == 0).map(|x| x * x).collect::<Vec<i32>>()

/home/igor > Sent to mshell (2940 bytes)

Received from GUI editor:

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

```rust >raw_text
fn main() {
   println!("Rust guarantees memory safety without garbage collection via ownership. \
      The borrow checker enforces references never outlive their data. \
      Stack allocation is preferred; heap uses Box, Rc, or Arc smart pointers. \
      The Drop trait ensures deterministic resource cleanup at end of scope.");
}
```

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

```rust &lt;raw_text &gt;sent1
use std::{env, fs};
fn main() {
    let raw = fs::read_to_string(env::var("MSH_VAR_raw_text").unwrap()).unwrap();
    let s: Vec<&str> = raw.trim().split('.').map(|s| s.trim()).filter(|s| !s.is_empty()).collect();
    println!("{}", s.get(0).unwrap_or(&""));
}
```

```rust &lt;raw_text &gt;sent2
use std::{env, fs};
fn main() {
    let raw = fs::read_to_string(env::var("MSH_VAR_raw_text").unwrap()).unwrap();
    let s: Vec<&str> = raw.trim().split('.').map(|s| s.trim()).filter(|s| !s.is_empty()).collect();
    println!("{}", s.iter().skip(1).take(2).cloned().collect::<Vec<_>>().join(". "));
}
```

```rust &lt;raw_text &gt;sent3
use std::{env, fs};
fn main() {
    let raw = fs::read_to_string(env::var("MSH_VAR_raw_text").unwrap()).unwrap();
    let s: Vec<&str> = raw.trim().split('.').map(|s| s.trim()).filter(|s| !s.is_empty()).collect();
    println!("{}", s.iter().skip(3).cloned().collect::<Vec<_>>().join(". "));
}
```

<!--@1 <sent1 >analysis1 async

The input is 1-2 sentences about Rust's memory model.

State the main Rust concept in 3 words max. Only the concept label.

-->

<!--@1 <sent2 >analysis2 async

The input is 1-2 sentences about Rust's memory model.

State the main Rust concept in 3 words max. Only the concept label.

-->

<!--@1 <sent3 >analysis3 async

The input is 1-2 sentences about Rust's memory model.

State the main Rust concept in 3 words max. Only the concept label.

-->

```bash await=analysis1,analysis2,analysis3
for p in "$(printenv MSH_VAR_analysis1)" "$(printenv MSH_VAR_analysis2)" "$(printenv MSH_VAR_analysis3)"; do
  i=0; while [ ! -s "$p" ] && [ $i -lt 180 ]; do sleep 1; i=$((i+1)); done
  if [ ! -s "$p" ]; then echo "[no response]" > "$p"; fi
done
```

<!--@merge-->

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

Three short Rust concept labels from different text chunks.

Synthesize them into one coherent sentence capturing Rust's memory management philosophy.

-->

```rust <analysis1 <analysis2 <analysis3 <summary
use std::{env, fs};
fn main() {
  println!("=== Map ===");
  for (label, var) in &[("Chunk 1","analysis1"),("Chunk 2","analysis2"),("Chunk 3","analysis3")] {
```

```
        let c = fs::read_to_string(env::var(&format!("MSH_VAR_{}", var)).unwrap()).unwrap();
        println!("  {}: {}", label, c.trim());
    }
    let summary = fs::read_to_string(env::var("MSH_VAR_summary").unwrap()).unwrap();
    println!("\n=== Reduce ===\n{}", summary.trim());
}
```

Rust guarantees memory safety without garbage collection via ownership. The borrow checker enforces references never outlive their data. Stack allocation is preferred; heap uses Box, Rc, or Arc smart pointers. The Drop trait ensures deterministic resource cleanup at end of scope.

[split] raw_text_1 = Rust guarantees memory safety without garbage collection via ownership. The borrow checker enforces references never outlive their data. Stack allocation is preferred; heap uses Box, Rc, or Arc smart pointers. The Drop trait ensures deterministic resource cleanup at end of scope.

Rust guarantees memory safety without garbage collection via ownership

The borrow checker enforces references never outlive their data. Stack allocation is preferred; heap uses Box, Rc, or Arc smart pointers

The Drop trait ensures deterministic resource cleanup at end of scope

[async llm] Launched PID 19301 \u2192 var=analysis1 (model @1)

[async llm] Launched PID 19302 \u2192 var=analysis2 (model @1)

[async llm] Launched PID 19304 \u2192 var=analysis3 (model @1)

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

[async] Waiting for PID 19301 (var=analysis1)...

[async] PID 19301 done (var=analysis1)

Ownership system

Rust\u2019s memory management philosophy uses the ownership system and `Drop` trait to guarantee memory safety without a garbage collector.

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

Chunk 1: Ownership system

Chunk 2: Memory safety

Chunk 3: Drop trait

=== Reduce ===

Rust\u2019s memory management philosophy uses the ownership system and `Drop` trait to guarantee memory safety without a garbage collector.

/home/igor > Sent to mshell (2486 bytes)

Received from GUI editor:

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

```rust >task
fn main() {
   println!("Write Rust fn median(data: &mut Vec<f64>) -> Option<f64> \
      that returns median (sorts in place), None for empty. \
      Include fn main() testing with [3.0,1.0,4.0,1.0,5.0] and empty vec.");
}
```

```rust >result
fn main() { println!("fail"); }
```

```rust >last_error
fn main() { println!("none"); }
```

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

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

The first input is a Rust coding task. The second is the previous compile error ("none" if first).

Return ONLY pure Rust code. No fences, no explanation. Must compile with rustc stable.

If there was a previous error, fix exactly that error.

~~~
-->

```rust <code

use std::{env, fs};

fn main() {

    println!("=== Generated ===\n{}\n",

        fs::read_to_string(env::var("MSH_VAR_code").unwrap()).unwrap().trim());

}

```

<!--@try-->

```rust <code

use std::{env, fs, process::Command};

fn main() {

    let code_path   = env::var("MSH_VAR_code").unwrap();

    let result_path = env::var("MSH_VAR_result").unwrap();

    let err_path    = "/tmp/msh_p21_err.txt";

    let src         = "/tmp/msh_p21_gen.rs";

    let bin         = "/tmp/msh_p21_bin";

    fs::copy(&code_path, src).unwrap();

    let c = Command::new("rustc").args([src, "-o", bin]).output().unwrap();

    if !c.status.success() {

        // write error to tmp file so CATCH can pass it to LLM

        let err = String::from_utf8_lossy(&c.stderr).to_string();

        fs::write(err_path, &err).unwrap();
~~~

````
        eprintln!("{}", err);
        std::process::exit(1);
    }
    let run = Command::new(bin).output().unwrap();
    print!("{}", String::from_utf8_lossy(&run.stdout));
    fs::write(&result_path, "ok\n").unwrap();
    println!("ok");
}
```
<!--@catch >last_error-->
```rust >result
use std::{env, fs};
fn main() {
    // pass compile error to last_error so LLM can fix it next iteration
    let err = fs::read_to_string("/tmp/msh_p21_err.txt")
        .unwrap_or_else(|_| "unknown error".into());
    fs::write(env::var("MSH_VAR_last_error").unwrap(), &err).unwrap();
    fs::write(env::var("MSH_VAR_result").unwrap(), "fail\n").unwrap();
    println!("fail");
}
```
<!--@end_try-->
<!--@end_loop-->
```rust <result
use std::{env, fs};
fn main() {
    println!("=== Final: {} ===",
````

```
    fs::read_to_string(env::var("MSH_VAR_result").unwrap()).unwrap().trim());
}
```

Write Rust fn median(data: &mut Vec<f64>) -> Option<f64> that returns median (sorts in place), None for empty. Include fn main() testing with [3.0,1.0,4.0,1.0,5.0] and empty vec.

fail

none

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

```
fn median(data: &mut Vec<f64>) -> Option<f64> {
  if data.is_empty() {
    return None;
  }
  data.sort_by(|a, b| a.partial_cmp(b).unwrap());
  let len = data.len();
  if len % 2 == 1 {
    Some(data[len / 2])
  } else {
    Some((data[len / 2 - 1] + data[len / 2]) / 2.0)
  }
}
fn main() {
  let mut test_data = vec![3.0, 1.0, 4.0, 1.0, 5.0];
  println!("Median: {:?}", median(&mut test_data));
  let mut empty_data = vec![];
  println!("Empty median: {:?}", median(&mut empty_data));
}
```

```
[try] executing try block
[try] try block succeeded
[loop] until check: last_line='fail' expected='ok'
[loop] Iteration 1/3 \u2014 condition not met, looping back
Median: Some(3.0)
Empty median: None
ok
fn median(data: &mut Vec<f64>) -> Option<f64> {
    if data.is_empty() {
        return None;
    }
    data.sort_by(|a, b| a.partial_cmp(b).unwrap());
    let len = data.len();
    if len % 2 == 1 {
        Some(data[len / 2])
    } else {
        Some((data[len / 2 - 1] + data[len / 2]) / 2.0)
    }
}
fn main() {
    let mut test_data = vec![3.0, 1.0, 4.0, 1.0, 5.0];
    println!("Median: {:?}", median(&mut test_data));
    let mut empty_data = vec![];
    println!("Empty median: {:?}", median(&mut empty_data));
}
[try] executing try block
Median: Some(3.0)
```

Empty median: None

ok

[try] try block succeeded

[loop] Exiting loop after 2 iteration(s). reason: until condition met

/home/igor > Median: Some(3.0)

Empty median: None

ok

/home/igor > Sent to mshell (1809 bytes)

Received from GUI editor:

Pattern 22 Multi-Variable Output: Structured Field Extraction

----

```rust >input
fn main() {
    println!("HashMap::entry(&mut self, key: K) -> Entry<'_, K, V> provides \
        in-place mutable access to an entry for key-based insertion or modification, \
        avoiding double-lookups via the Entry enum with occupied and vacant variants.");
}
```

<!--@1 <input >raw_response

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

SUMMARY: one sentence paraphrase

SIGNATURE: the function signature only

AUDIENCE: one word \u2014 beginner / intermediate / expert

-->

```rust <raw_response >summary >signature >audience
```

```
use std::{env, fs};

fn extract<'a>(text: &'a str, prefix: &str) -> &'a str {
    text.lines().find(|l| l.starts_with(prefix))
        .and_then(|l| l.strip_prefix(prefix))
        .map(|s| s.trim()).unwrap_or("n/a")
}

fn main() {
    let text = fs::read_to_string(env::var("MSH_VAR_raw_response").unwrap()).unwrap();
    let summary   = extract(&text, "SUMMARY: ");
    let signature = extract(&text, "SIGNATURE: ");
    let audience  = extract(&text, "AUDIENCE: ");
    fs::write(env::var("MSH_VAR_summary").unwrap(),   summary).unwrap();
    fs::write(env::var("MSH_VAR_signature").unwrap(), signature).unwrap();
    fs::write(env::var("MSH_VAR_audience").unwrap(),  audience).unwrap();
    println!("Summary   : {}", summary);
    println!("Signature : {}", signature);
    println!("Audience  : {}", audience);
}
```

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

Rewrite the summary for the stated audience level. One clear sentence.

-->

```rust <adaptation
use std::{env, fs};

fn main() {
    let c = fs::read_to_string(env::var("MSH_VAR_adaptation").unwrap()).unwrap();
    println!("\n=== Adapted for Audience ===\n{}", c.trim());
```

}

```

HashMap::entry(&mut self, key: K) -> Entry&lt;'_, K, V&gt; provides in-place mutable access to an entry for key-based insertion or modification, avoiding double-lookups via the Entry enum with occupied and vacant variants.

SUMMARY: HashMap::entry provides efficient key-based insertion or modification by returning an Entry enum that avoids multiple hash lookups.

SIGNATURE: HashMap::entry(&mut self, key: K) -> Entry&lt;'_, K, V&gt;

AUDIENCE: intermediate

Summary : HashMap::entry provides efficient key-based insertion or modification by returning an Entry enum that avoids multiple hash lookups.

Signature : HashMap::entry(&mut self, key: K) -> Entry&lt;'_, K, V&gt;

Audience : intermediate

`HashMap::entry` lets you efficiently insert or modify a value for a key by returning an `Entry` that ensures the hash lookup is done only once.

=== Adapted for Audience ===

`HashMap::entry` lets you efficiently insert or modify a value for a key by returning an `Entry` that ensures the hash lookup is done only once.

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

/home/igor > Sent to mshell (2599 bytes)

Received from GUI editor:

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

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

```config

subject=Rust async/await and the Future trait

target_audience=software engineer familiar with threads but new to async

quality_threshold=7

```
```

~~~
```rust >subject
fn main() { println!("Rust async/await and the Future trait"); }
```
```rust >target_audience
fn main() { println!("software engineer familiar with threads but new to async"); }
```
```rust >status
fn main() { println!("running"); }
```
```rust >iteration
fn main() { println!("0"); }
```
```rust >quality
fn main() { println!("0"); }
```
```rust >explanation
fn main() { println!(""); }
```
<!--@while status:running-->
```rust <iteration >iteration
use std::{env, fs};
fn main() {
    let path = env::var("MSH_VAR_iteration").unwrap();
    let val: u64 = fs::read_to_string(&path).unwrap_or_else(|_| "0".into())
        .trim().parse().unwrap_or(0);
    let new_val = val + 1;
    fs::write(&path, format!("{}", new_val)).unwrap();
~~~

~~~~
    print!("{}", new_val);
}
```
<!--@1 <subject <target_audience >explanation
The first input is a Rust subject. The second is the target audience.
Explain to that audience in exactly 3 sentences. No jargon they wouldn't know.
Use inline backtick code if it helps clarity.
-->
<!--@2 <explanation <target_audience >quality
Rate this explanation for the stated audience 1-10 (clarity, accuracy, engagement).
Reply with ONLY the integer, nothing else.
-->

```rust <iteration <quality >status
use std::{env, fs};
fn main() {
    let iter = fs::read_to_string(env::var("MSH_VAR_iteration").unwrap()).unwrap();
    let q    = fs::read_to_string(env::var("MSH_VAR_quality").unwrap()).unwrap();
    let sp   = env::var("MSH_VAR_status").unwrap();
    let qn: u64 = q.trim().parse().unwrap_or(0);
    println!("[Iter {}] Quality: {}/10", iter.trim(), qn);
    fs::write(&sp, if qn >= 7 { "done" } else { "running" }).unwrap();
}
```
<!--@end_while-->
<!--@3 <explanation >final_polish
Polish this Rust explanation for official documentation.
~~~~

Keep exactly 3 sentences. Use precise Rust terminology. No markdown.

-->

```rust &lt;final_polish &lt;quality &lt;iteration

use std::{env, fs};

fn main() {

    let polish  = fs::read_to_string(env::var("MSH_VAR_final_polish").unwrap()).unwrap();

    let quality = fs::read_to_string(env::var("MSH_VAR_quality").unwrap()).unwrap();

    let iter    = fs::read_to_string(env::var("MSH_VAR_iteration").unwrap()).unwrap();

    println!("=== Final (score={}/10, iterations={}) ===", quality.trim(), iter.trim());

    println!("{}", polish.trim());

}
```

[config] subject=Rust async/await and the Future trait

[config] target_audience=software engineer familiar with threads but new to async

[config] quality_threshold=7

Rust async/await and the Future trait

software engineer familiar with threads but new to async

running

0

0

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

1Rust's `async/await` lets you write code that looks synchronous but doesn't block threads while waiting for I/O operations like network requests or file reads, instead yielding control back to a runtime that can run other tasks. When you mark a function with `async fn`, it returns a `Future` - think of it as a state machine that represents work that hasn't completed yet, similar to how a thread represents ongoing work but much more lightweight. The `await` keyword is where the magic happens: it pauses your function at that point, lets other async tasks run, and resumes your function when the awaited operation completes, all without the overhead of creating new threads.

10

[Iter 1] Quality: 10/10

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

Rust's async/await syntax enables non-blocking I/O operations by yielding the current thread to the runtime during asynchronous waits, allowing concurrent execution of multiple tasks without thread blocking. An async function returns a Future, a state machine implementing the Future trait that represents asynchronous computation and is driven by repeated polling. The await operator suspends the current task until the Future is ready, resuming execution only when the operation completes, thereby achieving high concurrency with minimal resource overhead compared to thread-based models.

/home/igor > === Final (score=10/10, iterations=1) ===

Rust's async/await syntax enables non-blocking I/O operations by yielding the current thread to the runtime during asynchronous waits, allowing concurrent execution of multiple tasks without thread blocking. An async function returns a Future, a state machine implementing the Future trait that represents asynchronous computation and is driven by repeated polling. The await operator suspends the current task until the Future is ready, resuming execution only when the operation completes, thereby achieving high concurrency with minimal resource overhead compared to thread-based models.

/home/igor > Sent to mshell (1841 bytes)

Received from GUI editor:

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

----

~~~
```rust >items

fn main() {

   let s1 = r#"fn double(x: i32) -> i32 { x * 2 } fn main() { println!("{}", double(21)); }"#;

   let s2 = r#"fn main() { let v: Vec<i32> = (1..=5).map(|x| x*x).collect(); println!("{:?}", v); }"#;

   let s3 = r#"fn main() { let s = String::from("Ferris"); println!("Hello, {}!", s); }"#;

   let broken = r#"fn main( { let x = ; }"#;

   print!("{}\n{}\n{}\n{}", s1, s2, broken, s3);
~~~

````
}
```

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

<!--@try-->

```rust <item >parsed
use std::{env, fs, process::Command};
fn main() {
    let item_path   = env::var("MSH_VAR_item").unwrap();
    let parsed_path = env::var("MSH_VAR_parsed").unwrap();
    let snippet = fs::read_to_string(&item_path).unwrap();
    let src = "/tmp/mshell_p24_snippet.rs";
    let bin = "/tmp/mshell_p24_snippet_bin";
    fs::write(src, snippet.trim()).unwrap();
    let c = Command::new("rustc").args(["--edition","2021",src,"-o",bin]).output().unwrap();
    if !c.status.success() {
        eprintln!("{}", String::from_utf8_lossy(&c.stderr));
        std::process::exit(1);
    }
    fs::write(&parsed_path, snippet.trim()).unwrap();
}
```

<!--@1 <parsed >insight
The input is a short Rust snippet that compiled successfully.
In one sentence, describe what it does and what Rust feature it demonstrates.
-->

```rust <insight
use std::{env, fs};
````

fn main() {

    let c = fs::read_to_string(env::var("MSH_VAR_insight").unwrap()).unwrap();

    println!("[OK] {}", c.trim());

}

```

&lt;!--@catch &gt;parse_error--&gt;

```rust

fn main() {

    println!("[ERR] Compilation failed: try_block_failed");

}

```

<!--@end_try-->

<!--@end_foreach-->

```rust

fn main() {

    println!("=== Batch complete. Errors isolated, pipeline never stopped. ===");

}

```

fn double(x: i32) -> i32 { x * 2 } fn main() { println!("{}", double(21)); }

fn main() { let v: Vec<i32> = (1..=5).map(|x| x*x).collect(); println!("{:?}", v); }

fn main({ let x = ; }

fn main() { let s = String::from("Ferris"); println!("Hello, {}!", s); }[foreach] iter 1: item=fn double(x: i32) -> i32 { x * 2 } fn main() { println!("{}", double(21)); }

[try] executing try block

This code defines a simple function `double` that multiplies an integer by 2 and calls it from `main` to print 42, demonstrating Rust's basic function definition syntax and type annotations.
```

[OK] This code defines a simple function `double` that multiplies an integer by 2 and calls it from `main` to print 42, demonstrating Rust's basic function definition syntax and type annotations.

[try] try block succeeded

[foreach] iter 2: item=fn main() { let v: Vec<i32> = (1..=5).map(|x| x*x).collect(); println!("{:?}", v); }

[try] executing try block

This code creates a vector containing the squares of numbers 1 through 5 (resulting in [1, 4, 9, 16, 25]) and prints it, demonstrating Rust's iterator chains with range syntax, closures, and the `collect()` method for transforming iterators into collections.

[OK] This code creates a vector containing the squares of numbers 1 through 5 (resulting in [1, 4, 9, 16, 25]) and prints it, demonstrating Rust's iterator chains with range syntax, closures, and the `collect()` method for transforming iterators into collections.

[try] try block succeeded

[foreach] iter 3: item=fn main({ let x = ; }

[try] executing try block

```
error: this file contains an unclosed delimiter

 --> /tmp/mshell_p24_snippet.rs:1:23

  |

1 | fn main( { let x = ; }

  |        -             ^

  |        |

  |        unclosed delimiter

error: aborting due to 1 previous error
```

This code creates a vector containing the squares of numbers 1 through 5 (resulting in [1, 4, 9, 16, 25]) and prints it, demonstrating Rust's iterator chains with range syntax, closures, and the `collect()` method for transforming iterators into collections.

[try] try block failed, executing catch block

[ERR] Compilation failed: try_block_failed

[foreach] iter 4: item=fn main() { let s = String::from("Ferris"); println!("Hello, {}!", s); }

[try] executing try block

This code creates a heap-allocated `String` containing "Ferris" and prints "Hello, Ferris!", demonstrating Rust's `String` type for owned string data and string formatting with placeholders in `println!`.

[OK] This code creates a heap-allocated `String` containing "Ferris" and prints "Hello, Ferris!", demonstrating Rust's `String` type for owned string data and string formatting with placeholders in `println!`.

[try] try block succeeded

/home/igor > === Batch complete. Errors isolated, pipeline never stopped. ===

References:

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

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

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

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