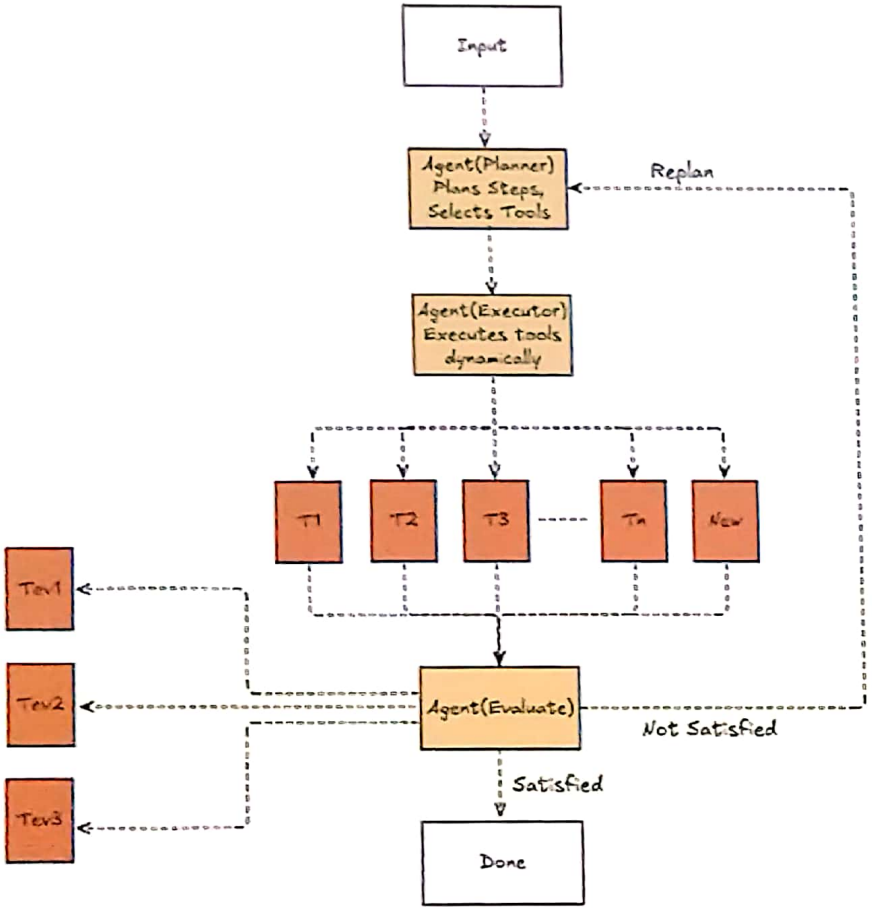
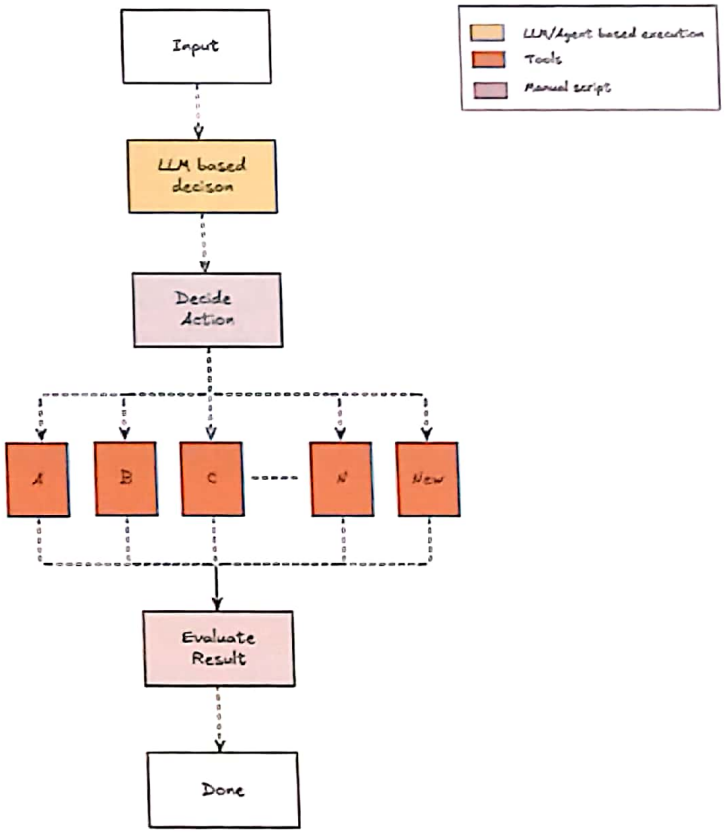
# Agents vs Workflows: A Turing Test for Your Architecture

Samir Kumar Nayak

# Agents vs LLM based fixed workflows



# Problem: Should i use an AI Agent or Simple Pipeline?

When building AI powered applications, we face this critical decision.

Wrong choice = wasted time, money or poor results!

Why not ask ChatGPT?

ChatGPT says:

- "It depends"...
- Gives an opinion which might change each time
- Black box reasoning, no metrics how the opinion is formed

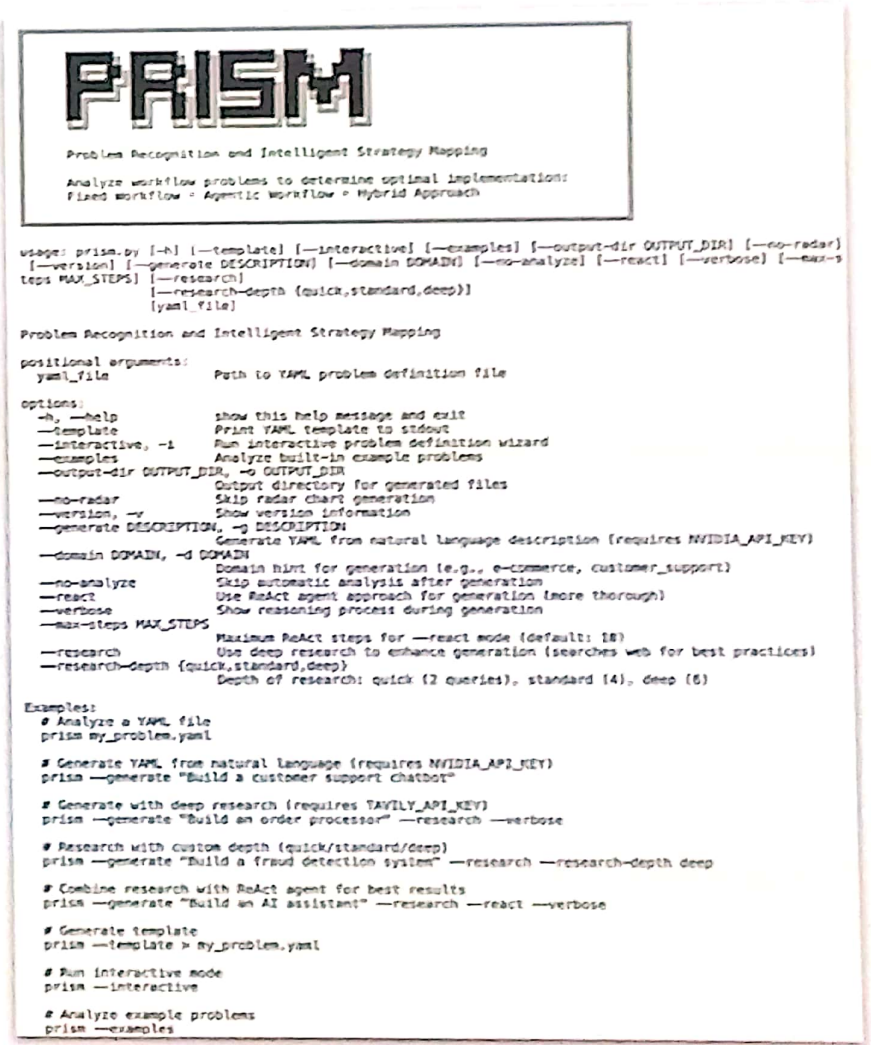
# The Turing Test Framework: PRISM

**PRISM**: Problem Recognition and Intelligent Strategy Mapping

A framework that analyzes workflow problems and recommends whether to implement your LLM based workflow as a:

- Fixed Workflow
- Agentic Workflow
- Hybrid Approach

The framework models your problem as a graph of nodes (processing steps) and edges (transitions), then analyzes structural properties to provide a recommendation with confidence scores.



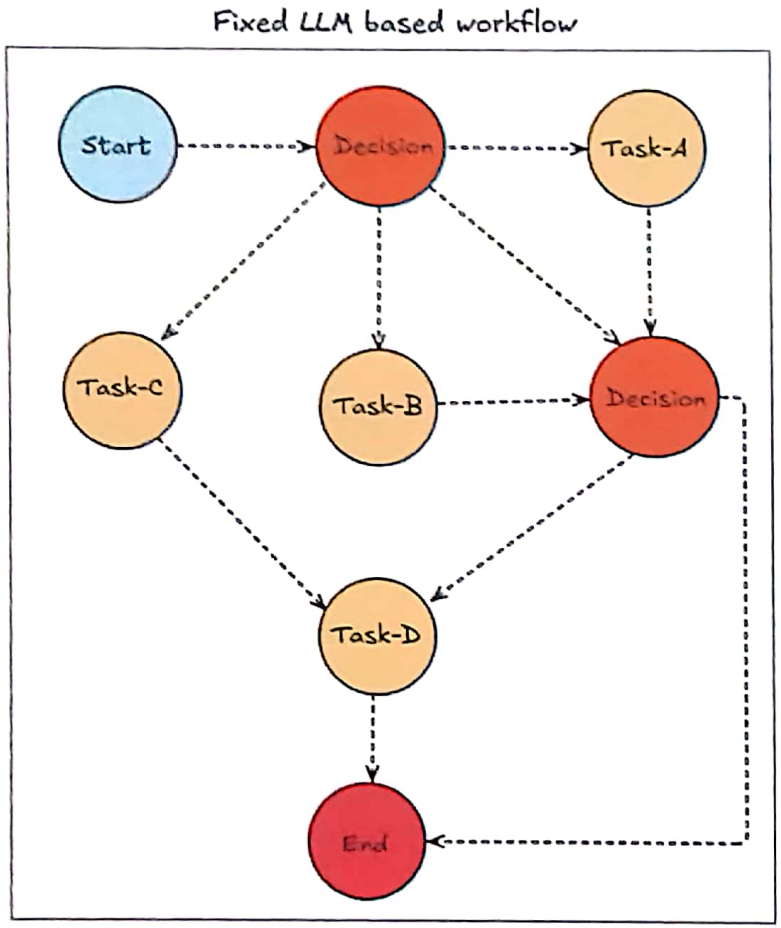
```
PRISM

Problem Recognition and Intelligent Strategy Mapping

Analyze workflow problems to determine optimal implementation:
Fixed workflow • Agentic workflow • Hybrid Approach

usage: prism.py [-h] [--template] [--interactive] [--examples] [--output-dir OUTPUT_DIR] [--no-radar]
  [--version] [--generate DESCRIPTION] [--domain DOMAIN] [--no-analyze] [--react] [--verbose] [--max-s
teps MAX_STEPS] [--research]
                [--research-depth {quick,standard,deep}]
                [yaml_file]

Problem Recognition and Intelligent Strategy Mapping

positional arguments:
  yaml_file             Path to YAML problem definition file

options:
  -h, --help            show this help message and exit
  --template            Print YAML template to stdout
  --interactive, -i     Run interactive problem definition wizard
  --examples            Analyze built-in example problems
  --output-dir OUTPUT_DIR, -o OUTPUT_DIR
                        Output directory for generated files
  --no-radar            Skip radar chart generation
  --version, -v         Show version information
  --generate DESCRIPTION, -g DESCRIPTION
                        Generate YAML from natural language description (requires NVIDIA_API_KEY)
  --domain DOMAIN, -d DOMAIN
                        Domain hint for generation (e.g., e-commerce, customer_support)
  --no-analyze          Skip automatic analysis after generation
  --react               Use ReAct agent approach for generation (more thorough)
  --verbose             Show reasoning process during generation
  --max-steps MAX_STEPS
                        Maximum ReAct steps for --react mode (default: 10)
  --research            Use deep research to enhance generation (searches web for best practices)
  --research-depth {quick,standard,deep}
                        Depth of research: quick (2 queries), standard (4), deep (6)

Examples:
  # Analyze a YAML file
  prism my_problem.yaml

  # Generate YAML from natural language (requires NVIDIA_API_KEY)
  prism --generate "Build a customer support chatbot"

  # Generate with deep research (requires TAVILY_API_KEY)
  prism --generate "Build an order processor" --research --verbose

  # Research with custom depth (quick/standard/deep)
  prism --generate "Build a fraud detection system" --research --research-depth deep

  # Combine research with ReAct agent for best results
  prism --generate "Build an AI assistant" --research --react --verbose

  # Generate template
  prism --template > my_problem.yaml

  # Run interactive mode
  prism --interactive

  # Analyze example problems
  prism --examples
```

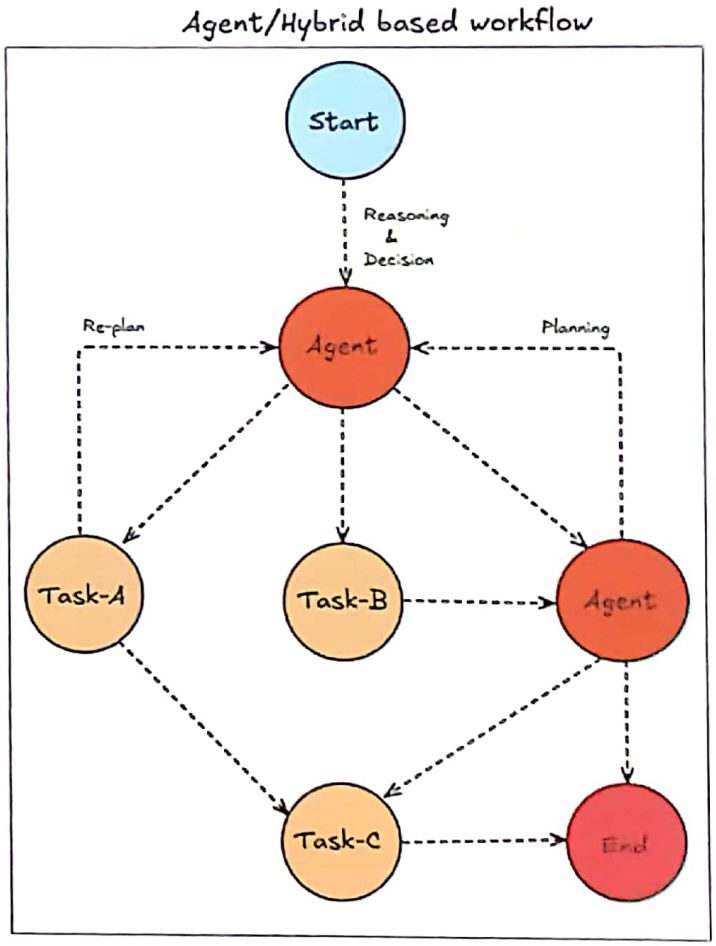
# The Problem-First Approach

Before writing a single line of code:

- Map the problem as a graph
- Problem = Series of States(Nodes) + Transitions(Edges)
- Architecture is defined by how we move between nodes
- The right choice depends on: Who decides the transition?

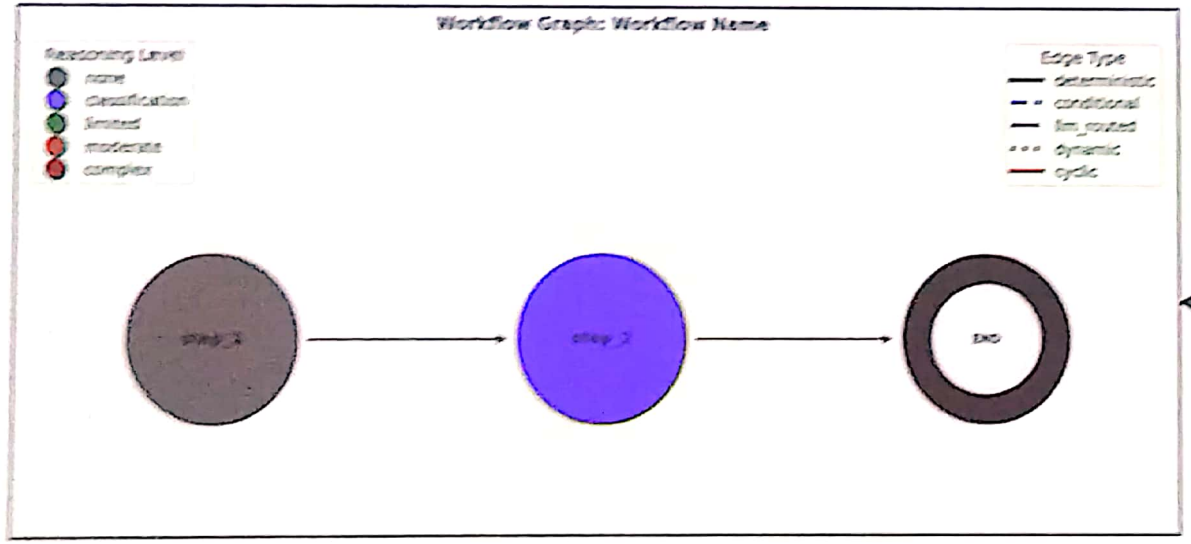
# Fixed & Agent based workflow (Problem breakdown)



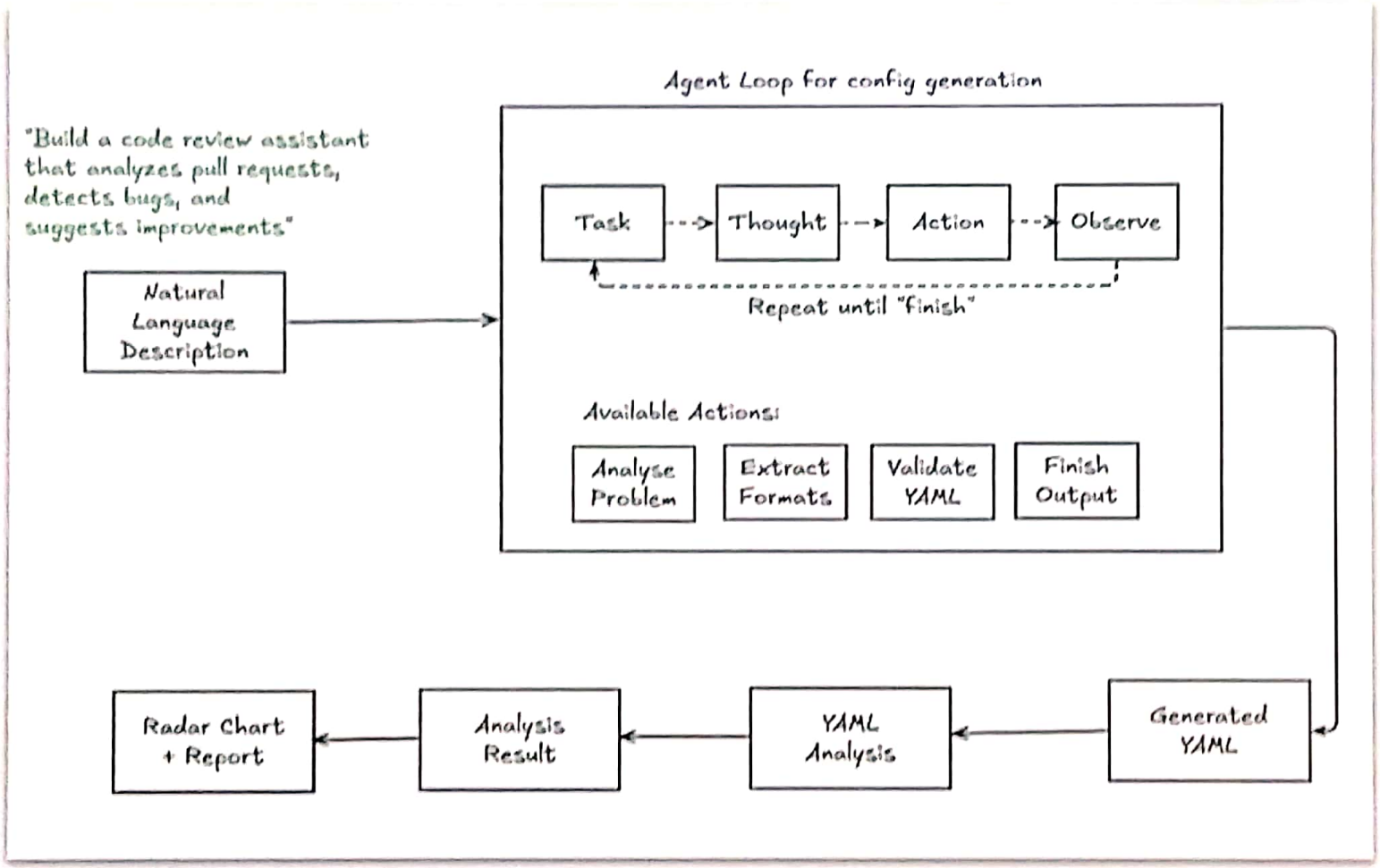
# Structure the problem statement

| Section | Purpose |
|---|---|
| Problem | Name, description, and domain classification |
| Constraints | Latency limits, determinism, auditability requirements |
| Steps | Processing nodes with input/output formats and reasoning levels |
| Edges | Connections between steps with transition types |

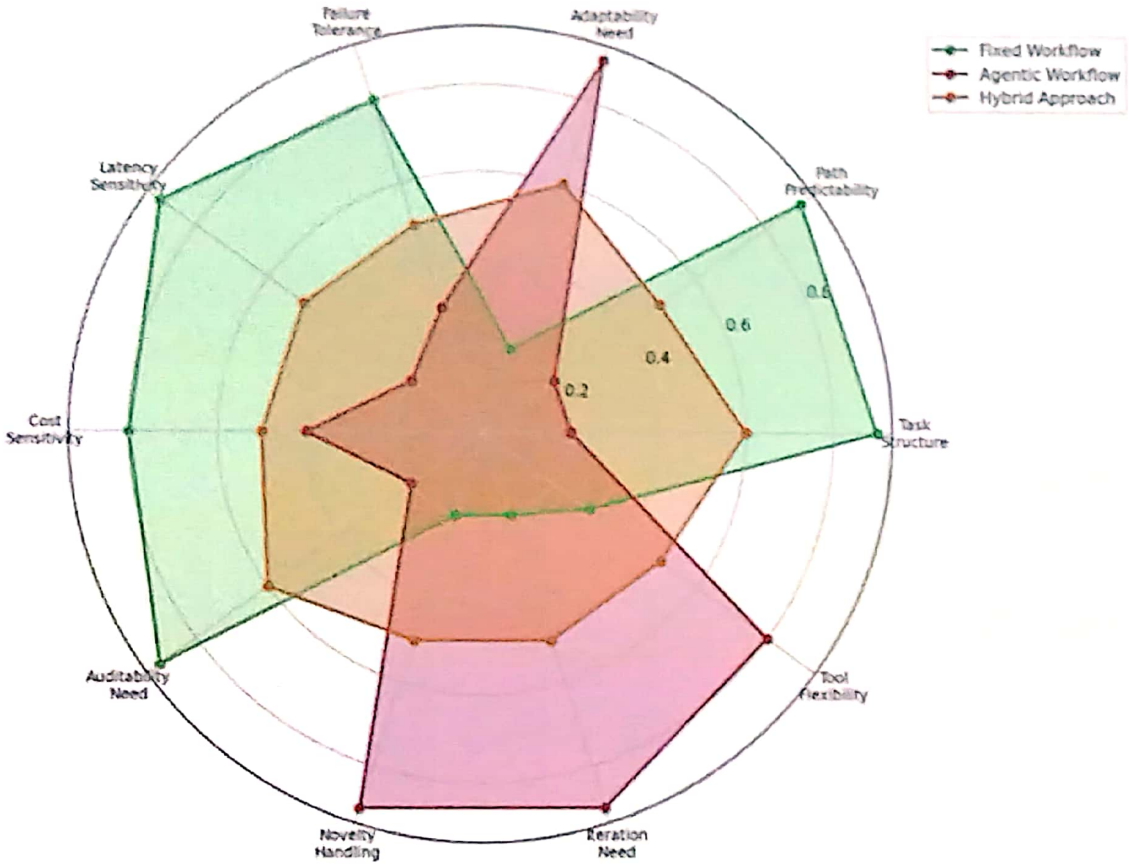



```yaml
# PRISM Workflow Template

problem:
  name: "My Workflow"
  description: "What this workflow does"

constraints:
  latency_ms: null              # null | <milliseconds>
  requires_determinism: false   # true | false
  requires_auditability: false  # true | false

steps:
  - name: "step_1"
    description: "First step"
    input_format: "structured"   # structured | semi | unstructured
    output_format: "structured"  # structured | semi | unstructured
    reasoning: "none"            # none | classification | limited | moderate | complex
    requires_retry: false        # true | false
    requires_self_eval: false    # true | false

  - name: "step_2"
    description: "Second step"
    input_format: "structured"
    output_format: "structured"
    reasoning: "classification"
    requires_retry: false
    requires_self_eval: false

edges:
  - from: "step_1"
    to: "step_2"                 # <step_name> | END
    type: "deterministic"        # deterministic | conditional | llm_routed | dynamic | cyclic
    conditions: []               # for conditional edges
    max_iterations: null         # for cyclic edges: null | <number>

  - from: "step_2"
    to: "END"
    type: "deterministic"
```

# PRISM Workflow



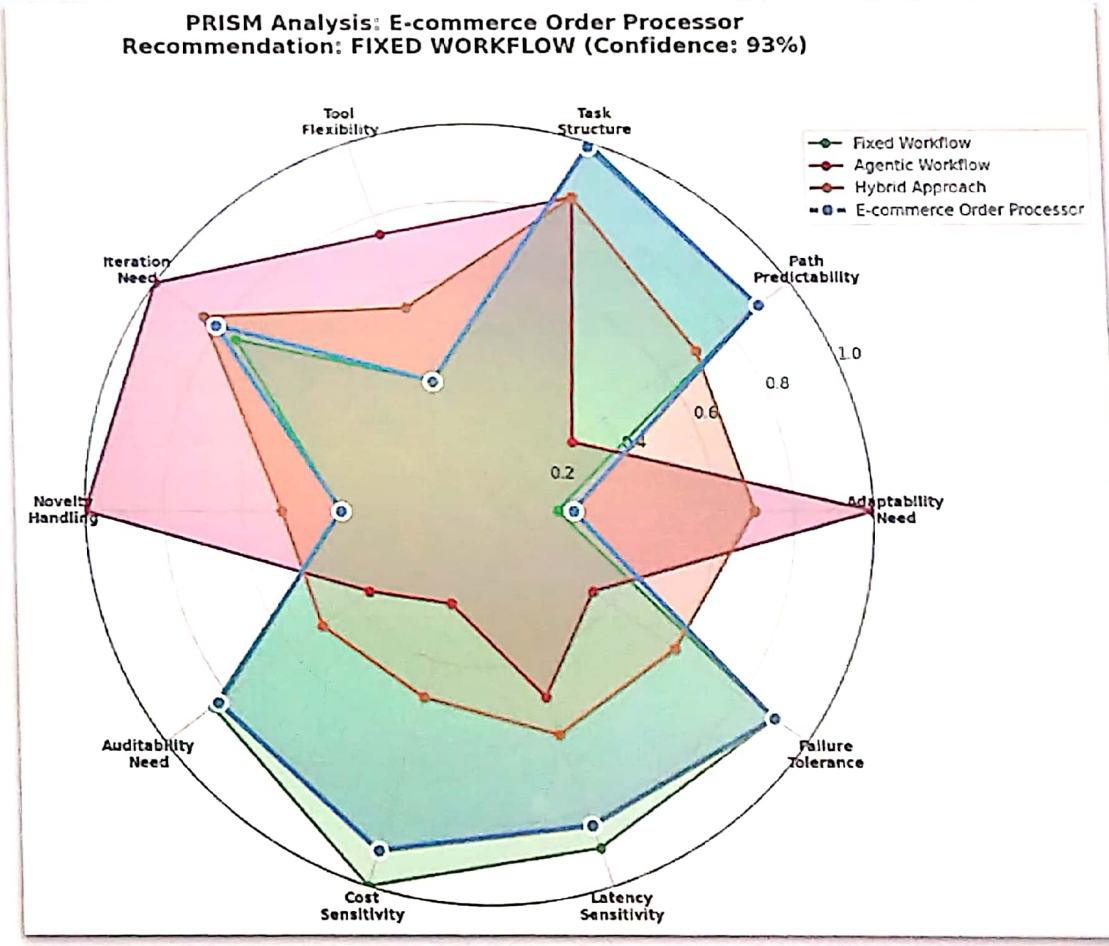
# Workflow dimension profiles



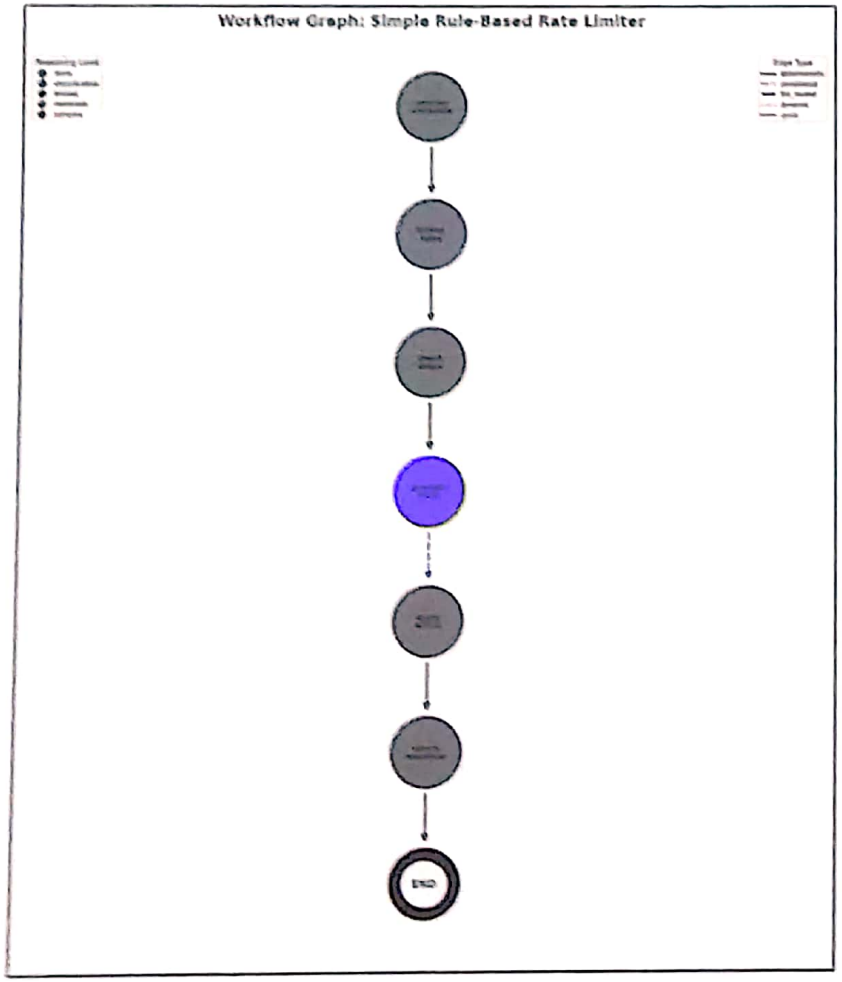

| # | Dimension | What It Measures | Favors |
|---|---|---|---|
| 1 | **Adaptability Need** | Need to handle changing inputs | Agentic |
| 2 | **Path Predictability** | How deterministic the execution path is | Fixed |
| 3 | **Task Structure** | How well-defined the requirements are | Both |
| 4 | **Tool Flexibility** | Benefit from iterative tool selection | Agentic |
| 5 | **Iteration Need** | Need for loops and retries | Agentic |
| 6 | **Novelty Handling** | Handle unexpected/edge-case scenarios | Agentic |
| 7 | **Auditability Need** | Tracing and compliance requirements | Fixed |
| 8 | **Cost Sensitivity** | Importance of minimizing API costs | Fixed |
| 9 | **Latency Sensitivity** | Importance of fast response time | Fixed |
| 10 | **Failure Tolerance** | Need for graceful error handling | Fixed |

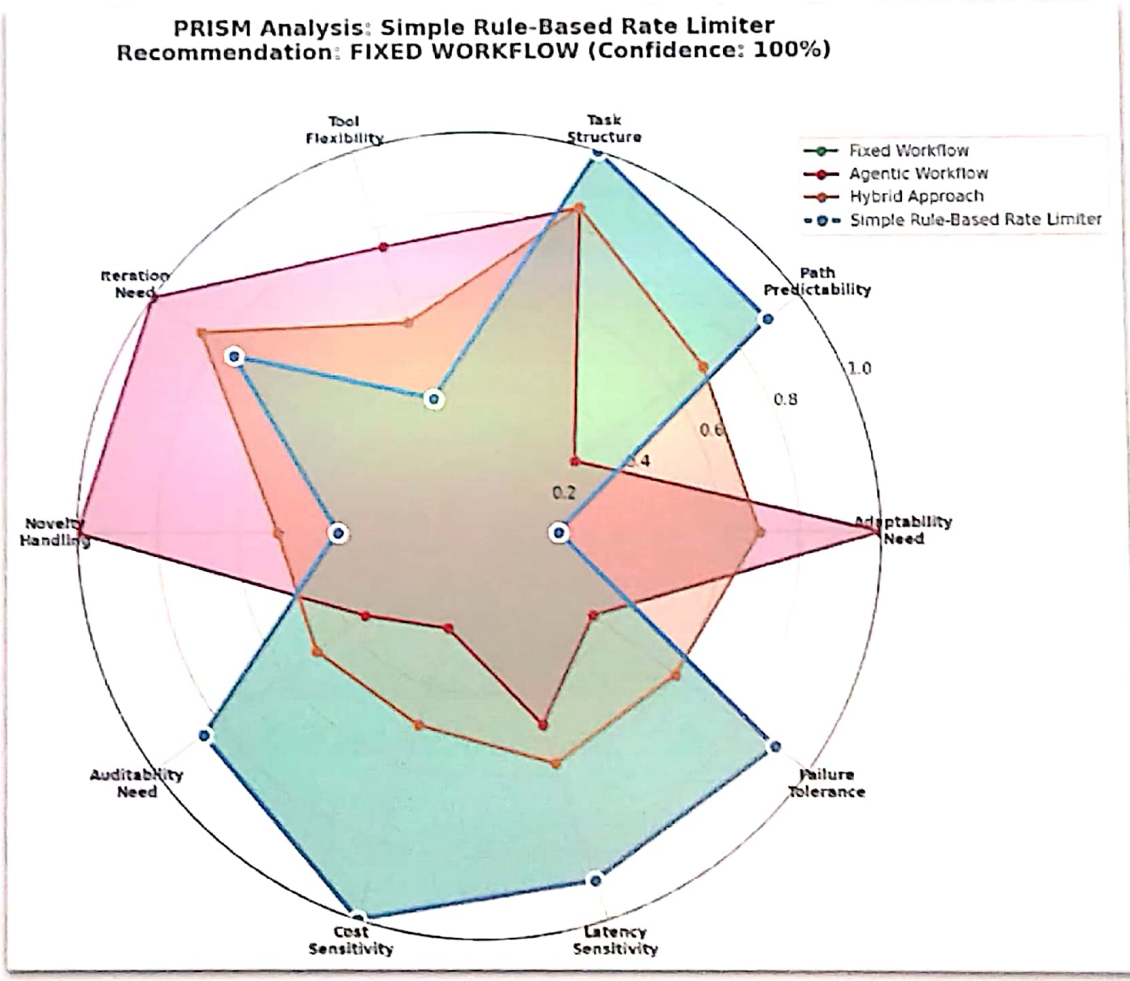
# Examples: e-commerce order processing

```yaml
# PRISM: E-commerce Order Processing Workflow
problem:
  name: "E-commerce Order Processor"
  description: "Processes incoming orders from validation to fulfillment"
  domain: "e-commerce"

constraints:
  latency_ms: 2000
  requires_determinism: true
  requires_auditability: true

steps:
  - {name: validate_order, input_format: semi, output_format: structured, reasoning: classification,
  requires_retry: false, requires_self_eval: false}
  - {name: check_inventory, input_format: structured, output_format: structured, reasoning: none,
  requires_retry: false, requires_self_eval: false}
  - {name: calculate_pricing, input_format: structured, output_format: structured, reasoning: limited,
  requires_retry: false, requires_self_eval: false}
  - {name: process_payment, input_format: structured, output_format: structured, reasoning: none,
  requires_retry: true, requires_self_eval: false}
  - {name: handle_payment_failure, input_format: structured, output_format: structured, reasoning: limited,
  requires_retry: true, requires_self_eval: false}
  - {name: create_shipment, input_format: structured, output_format: structured, reasoning: none,
  requires_retry: false, requires_self_eval: false}
  - {name: send_confirmation, input_format: structured, output_format: structured, reasoning: limited,
  requires_retry: false, requires_self_eval: false}

edges:
  - {from: validate_order, to: check_inventory, type: conditional, conditions: [order_valid=true]}
  - {from: validate_order, to: END, type: conditional, conditions: [order_valid=false]}
  - {from: check_inventory, to: calculate_pricing, type: deterministic}
  - {from: calculate_pricing, to: process_payment, type: deterministic}
  - {from: process_payment, to: create_shipment, type: conditional, conditions: [payment_success=true]}
  - {from: process_payment, to: handle_payment_failure, type: conditional, conditions:
  [payment_success=false]}
  - {from: handle_payment_failure, to: process_payment, type: cyclic, max_iterations: 3}
  - {from: handle_payment_failure, to: END, type: conditional, conditions: [max_retries_exceeded]}
  - {from: create_shipment, to: send_confirmation, type: deterministic}
  - {from: send_confirmation, to: END, type: deterministic}
```

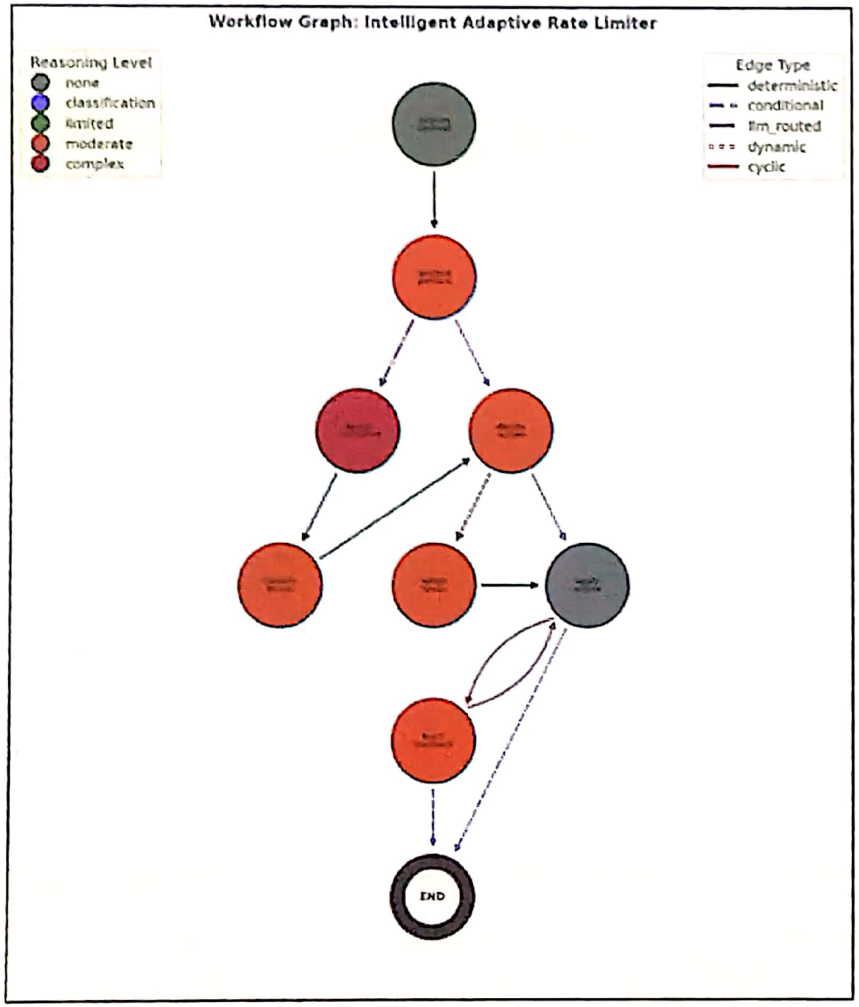


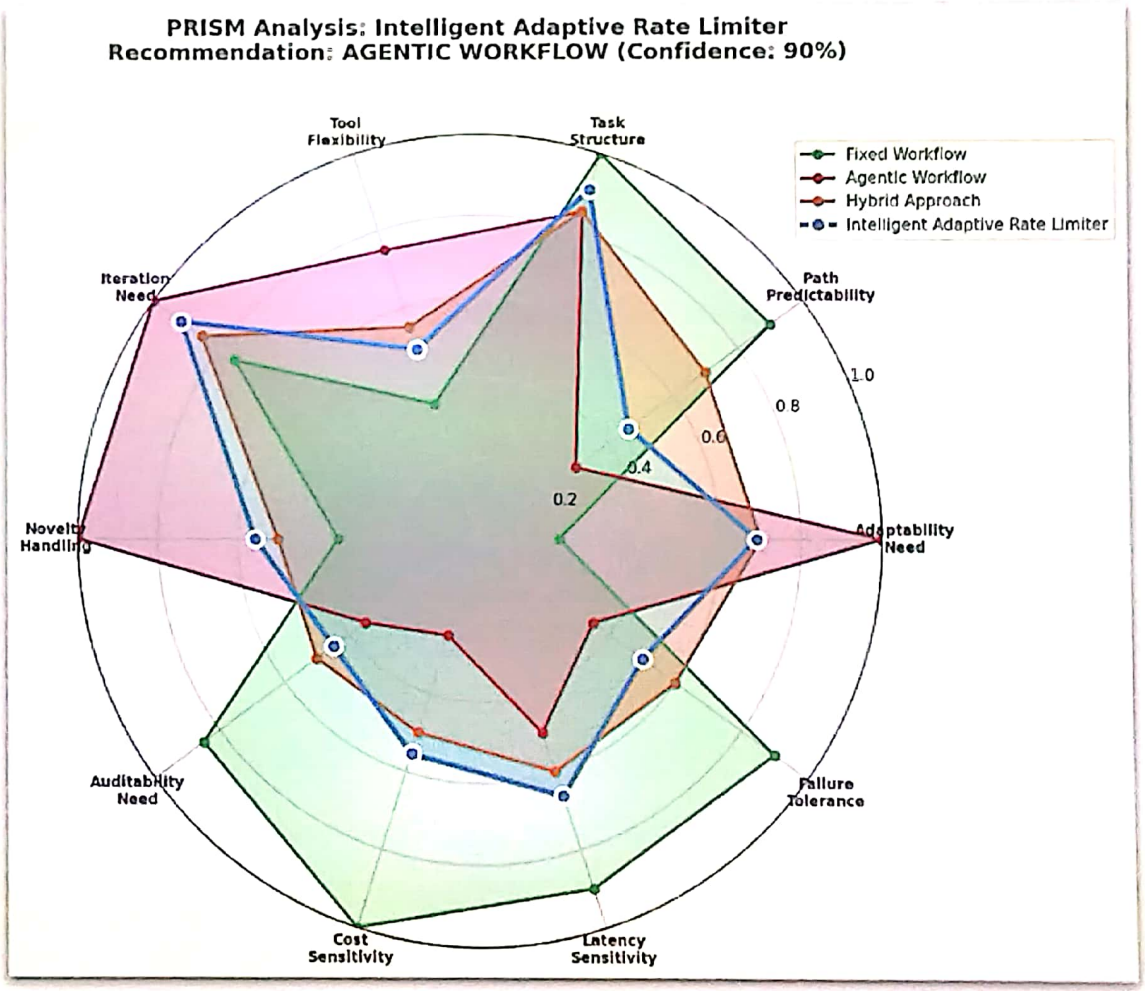
# Examples: Simple rate limiter for custom API



# Examples: Intelligent rate limiter for custom API



# Dimension Score Calculation from YAML

Formula: score = base + factor × (weighted sum of inputs)

| Dimension | Favors | Base | Factor | Range | Weight Components |
|---|---|---|---|---|---|
| Adaptability Need | Agentic | 0.2 | 0.8 | 0.2–1.0 | inverse_data_format: 0.35, inverse_reasoning: 0.35, dynamic_edges: 0.15, self_eval_nodes: 0.15 |
| Path Predictability | Fixed | 0.3 | 0.6 | 0.3–0.9 | paths_enumerable: 0.4, inverse_dynamic_edges: 0.3, no_unbounded_cycles: 0.3 |
| Task Structure | Both | 0.85 | 0.15 | 0.85–1.0 | data_format: 0.5, reasoning: 0.3, inverse_dynamic_edges: 0.2 |
| Tool Flexibility | Agentic | 0.35 | 0.4 | 0.35–0.75 | llm_routed_edges: 0.3, dynamic_edges: 0.3, inverse_reasoning: 0.2, complex_nodes: 0.2 |
| Iteration Need | Agentic | 0.75 | 0.25 | 0.75–1.0 | has_cycles: 0.25, unbounded_cycles: 0.25, retry_nodes: 0.25, self_eval_nodes: 0.25 |
| Novelty Handling | Agentic | 0.35 | 0.65 | 0.35–1.0 | unstructured_ratio: 0.35, inverse_reasoning: 0.35, exploration: 0.15, dynamic_edges: 0.15 |
| Auditability Need | Fixed | 0.35 | 0.5 | 0.35–0.85 | paths_enumerable: 0.35, data_format: 0.25, topology: 0.25, inverse_dynamic_edges: 0.15 |
| Cost Sensitivity | Fixed | 0.25 | 0.75 | 0.25–1.0 | reasoning: 0.35, no_cycles: 0.25, inverse_complex_nodes: 0.25, inverse_retry_nodes: 0.15 |
| Latency Sensitivity | Fixed | 0.5 | 0.4 | 0.5–0.9 | reasoning: 0.3, no_cycles: 0.3, inverse_dynamic_edges: 0.2, topology: 0.2 |
| Failure Tolerance | Fixed | 0.35 | 0.55 | 0.35–0.9 | paths_enumerable: 0.3, no_unbounded_cycles: 0.25, no_complex_reasoning: 0.25, topology: 0.2 |

*ll weights are heuristically designed, not empirically validated(weights)*

# Intermediate Metrics

| Metric | Source YAML Fields | Formula |
|---|---|---|
| data_score | steps[].input_format, steps[].output_format | Count of steps where both are "structured" ÷ total steps |
| reasoning_score | steps[].reasoning | Count of steps with "none", "classification", or "limited" ÷ total steps |
| dynamic_edges | edges[].type == "dynamic" | Count |
| llm_routed_edges | edges[].type == "llm_routed" | Count |
| has_cycles | edges[].type == "cyclic" | true if any exist |
| unbounded_cycles | edges[].type == "cyclic" + edges[].max_iterations == null | true if any cyclic edge has no limit |
| paths_enumerable | edges[].type, edges[].max_iterations | true if no dynamic edges AND no unbounded cycles |
| complex_nodes | steps[].reasoning in ["moderate", "complex"] | Count |
| retry_nodes | steps[].requires_retry == true | Count |
| self_eval_nodes | steps[].requires_self_eval == true | Count |
| unstructured_ratio | steps[].input_format, steps[].output_format | (unstructured inputs + outputs) ÷ (total_nodes × 2) |

# PRISM: Key Takeaways

The problem we are trying to solve:

**"Should I build this as a fixed workflow or let an agent figure it out?"**

This question costs teams weeks of trial-and-error, over-engineered solutions, and missed performance targets.

The Core Insight:

- Not every LLM workflow needs to be agentic.
- Not every workflow can be fixed.
- The problem structure determines the right approach.

PRISM Approach:

- Model: Represent any workflow as a structured graph of steps and transitions
- Measure: Score in 10 workflow dimension metrics that differentiate Fixed from Agentic
- Decide: Get a quantified recommendation, not a guess

*The dimension weights are heuristically designed based on domain expertise and logical relationships between workflow characteristics. They follow consistent patterns (primary vs secondary factors) and prevent edge-case scores. Future work could empirically validate these weights against real-world implementation outcomes.