Routing Logic Specification (Prolog Predicates)

Sovereign AI Infrastructure: Declarative Routing & Classification Rules

Document Version: 1.0
Date: February 5, 2026
Status: Draft for Review
Owner: Logic Systems Engineer / Routing Lead

Document Control

Version	Date	Author	Changes
1.0	2026-02-05	Routing Architecture Team	Initial draft

Reviewers:
- [ ] Technical Lead
- [ ] ML Lead
- [ ] Backend Lead

Dependencies:
- System Architecture Document (SAD) v1.0
- Product Requirements Document (PRD) v1.0
- Data Architecture & Memory Design Document v1.0

Table of Contents

---

1. Executive Summary

1.1 Purpose

This document defines the **declarative routing logic** for the Sovereign AI Infrastructure using **SWI-Prolog**. The routing system is the "decision-making brain" that analyzes user requests and determines:

1. **Domain**: What kind of task is this? (coding, reasoning, creative, documentation)
2. **Stakes**: How critical is correctness? (low, medium, high)
3. **Model**: Which specialist Worker should handle it?
4. **Validation**: What level of validation is needed?
5. **Tools**: Which multimodal pipelines are required? (OCR, vision, embeddings)

1.2 Why Prolog?

Advantages over Imperative Code (Python if-else):
- **Declarative**: Rules express "what" not "how" → easier to understand and modify
- **Inspectable**: All routing logic is visible, auditable
- **Maintainable**: Add new rules without touching existing code
- **Backtracking**: Natural handling of multiple matching rules (try alternatives)
- **Pattern Matching**: Powerful for complex request classification

Example Comparison:

Python (Imperative):

```python
def classify_domain(request):
    if "refactor" in request and "architecture" in request:
        return "coding_architecture"
    elif "implement" in request or "write function" in request:
        return "coding_implementation"
    # ... 50 more if-elif branches
```

Prolog (Declarative):

```prolog
classify_domain(Request, coding_architecture) :-
    contains_keywords(Request, [refactor, architecture]).

classify_domain(Request, coding_implementation) :-
    contains_keywords(Request, [implement, 'write function']).
```

1.3 Routing Decision Output

JSON Structure (returned to Python orchestrator):

```json
{
  "domain": "coding_architecture",
  "stakes": "high",
  "recommended_model": "qwen_coder_32b",
  "validation_policy": "block_by_block",
  "tools_required": ["embeddings"],
  "confidence": 0.92,
  "reasoning": "Keywords indicate architectural refactoring; high complexity suggests
high stakes; Qwen Coder best for architecture",
  "alternatives": [
    {"model": "gpt_oss_20b", "confidence": 0.67}
  ]
}
```

---

2. Prolog Architecture Overview

2.1 Prolog Knowledge Base Structure

File Organization:

```
config/routing_rules.pl
├── 1. Facts & Constants
│   ├── model_capabilities.pl    (What each model is good at)
│   ├── domain_keywords.pl       (Keywords for each domain)
│   └── stakes_indicators.pl     (Complexity/risk indicators)
├── 2. Core Rules
│   ├── domain_classification.pl (Domain detection)
│   ├── stakes_assessment.pl     (Stakes evaluation)
│   ├── model_selection.pl       (Model recommendation)
│   ├── validation_policy.pl     (Validation rigor)
│   └── tool_detection.pl        (Multimodal tool requirements)
└── 3. Utilities
    ├── text_analysis.pl         (Keyword matching, NLP-lite)
    ├── confidence_scoring.pl    (Confidence calculation)
    └── fallback_rules.pl        (Uncertainty handling)
```

2.2 Execution Flow

```
User Request (Text)
    ↓
[1. Tokenize & Normalize]
    ↓
[2. Domain Classification] → domain(Domain)
    ↓
[3. Stakes Assessment] → stakes(Stakes)
    ↓
[4. Model Selection] → model(Model, Confidence)
    ↓
[5. Validation Policy] → validation_policy(Policy)
    ↓
[6. Tool Detection] → tools_required(ToolsList)
    ↓
[7. Confidence Scoring] → overall_confidence(Score)
    ↓
[8. Reasoning Generation] → reasoning(Explanation)
    ↓
JSON Output → Return to Python
```

2.3 Integration with Router Model

Hybrid Approach:
1. **Router Model** (Granite-Micro 3B): Initial classification (fast, neural)
2. **Prolog Rules**: Refine classification, apply constraints, select model
3. **Embedding Fallback**: If Prolog confidence low, use semantic similarity

Why Hybrid?:
- Neural model: Good at capturing semantic nuances, patterns in language
- Prolog rules: Good at enforcing hard constraints, logical reasoning
- Embeddings: Good at handling novel/ambiguous requests (fallback)

---

3. Core Routing Predicates

3.1 Main Entry Point

```
%% route(+Request, -RoutingDecision)
%
% Main routing predicate. Analyzes request and produces routing decision.
%
% @param Request: Atom or string (user's task description)
% @param RoutingDecision: dict containing domain, stakes, model, etc.
%
% Example:
%   ?- route("Refactor payment module for SOLID principles", Decision).
%   Decision = {domain: coding_architecture, stakes: high, model: qwen_coder, ...}

route(Request, Decision) :-
    % Step 1: Normalize request
    normalize_text(Request, NormalizedRequest),

    % Step 2: Classify domain
    classify_domain(NormalizedRequest, Domain),

    % Step 3: Assess stakes
    assess_stakes(NormalizedRequest, Domain, Stakes),

    % Step 4: Select model
    select_model(Domain, Stakes, Model, ModelConfidence),

    % Step 5: Determine validation policy
    validation_policy(Stakes, Policy),

    % Step 6: Detect tool requirements
    detect_tools(NormalizedRequest, ToolsList),

    % Step 7: Overall confidence
    overall_confidence(Domain, Stakes, Model, ModelConfidence, Confidence),

    % Step 8: Generate reasoning
    generate_reasoning(Domain, Stakes, Model, Policy, Reasoning),

    % Step 9: Package decision
    Decision = _{
        domain: Domain,
        stakes: Stakes,
        recommended_model: Model,
        validation_policy: Policy,
        tools_required: ToolsList,
        confidence: Confidence,
        reasoning: Reasoning
    }.
```

3.2 Utility Predicates

```prolog
%% normalize_text(+RawText, -Normalized)
% Convert to lowercase, strip punctuation, tokenize
normalize_text(RawText, Normalized) :-
    downcase_atom(RawText, Lower),
    split_string(Lower, " .,!?;:", " ", Tokens),
    atomic_list_concat(Tokens, ' ', Normalized).

%% contains_keywords(+Text, +Keywords)
% Check if text contains any of the keywords
contains_keywords(Text, Keywords) :-
    member(Keyword, Keywords),
    sub_atom(Text, _, _, _, Keyword).

%% contains_all_keywords(+Text, +Keywords)
% Check if text contains ALL keywords
contains_all_keywords(_, []).
contains_all_keywords(Text, [Keyword|Rest]) :-
    sub_atom(Text, _, _, _, Keyword),
    contains_all_keywords(Text, Rest).

%% keyword_count(+Text, +Keywords, -Count)
% Count how many keywords from list appear in text
keyword_count(Text, Keywords, Count) :-
    findall(K, (member(K, Keywords), sub_atom(Text, _, _, _, K)), Matches),
    length(Matches, Count).
```

4. Domain Classification Rules

4.1 Domain Definitions

```prolog
%% Domain types
domain_type(coding_architecture).
domain_type(coding_implementation).
domain_type(reasoning).
domain_type(creative).
domain_type(documentation).
domain_type(unknown).  % Fallback
```

4.2 Domain-Specific Keywords

```prolog
%% domain_keywords(?Domain, ?Keywords)
% Define keywords associated with each domain

% Coding - Architecture
domain_keywords(coding_architecture, [
    refactor, architecture, design, pattern, solid, dependency, interface,
    abstraction, coupling, cohesion, 'design pattern', microservice,
    'system design', modular, scalable, extensible, 'class diagram'
]).

% Coding - Implementation
domain_keywords(coding_implementation, [
    implement, 'write function', 'write class', create, build, develop,
    optimize, performance, 'bug fix', debug, test, unittest, algorithm,
    'data structure', loop, recursion, 'write code', script
]).

% Reasoning & Planning
domain_keywords(reasoning, [
    analyze, evaluate, assess, plan, strategy, decide, compare, explain,
    reason, logic, problem, solution, approach, 'pros and cons',
    tradeoff, recommendation, 'what if', scenario
]).

% Creative Writing
domain_keywords(creative, [
    write, story, narrative, blog, article, essay, creative, poem,
    draft, tone, style, engaging, compelling, marketing, copy,
    announcement, 'social media', tweet
]).

% Documentation
domain_keywords(documentation, [
    document, readme, manual, guide, tutorial, explain, describe,
    'how to', overview, summary, report, specification, api,
    documentation, 'user guide'
]).
```

4.3 Domain Classification Logic

```prolog
%% classify_domain(+Request, -Domain)
% Classify request into a domain based on keywords and patterns

% Rule 1: Coding Architecture
classify_domain(Request, coding_architecture) :-
    domain_keywords(coding_architecture, ArchKeywords),
    keyword_count(Request, ArchKeywords, Count),
    Count >= 2,  % At least 2 architecture keywords
    !.  % Cut: prevent backtracking if this succeeds

% Rule 2: Coding Implementation
classify_domain(Request, coding_implementation) :-
    domain_keywords(coding_implementation, ImplKeywords),
    keyword_count(Request, ImplKeywords, Count),
    Count >= 2,
    !.

% Rule 3: Coding (generic - if any coding keyword but not architecture)
classify_domain(Request, coding_implementation) :-
    (contains_keywords(Request, [code, function, class, method, program]) ;
     contains_keywords(Request, [python, java, javascript, cpp, rust])),
    \+ classify_domain(Request, coding_architecture),  % Not architecture
    !.

% Rule 4: Reasoning
classify_domain(Request, reasoning) :-
    domain_keywords(reasoning, ReasonKeywords),
    keyword_count(Request, ReasonKeywords, Count),
    Count >= 2,
    !.

% Rule 5: Creative
classify_domain(Request, creative) :-
    domain_keywords(creative, CreativeKeywords),
    keyword_count(Request, CreativeKeywords, Count),
    Count >= 2,
    !.

% Rule 6: Documentation
classify_domain(Request, documentation) :-
    domain_keywords(documentation, DocKeywords),
    keyword_count(Request, DocKeywords, Count),
    Count >= 2,
    !.

% Rule 7: Fallback - Unknown
classify_domain(_, unknown) :-
    !.  % Catch-all
```

4.4 Domain Classification Confidence

```
%% domain_confidence(+Request, +Domain, -Confidence)
% Calculate confidence score for domain classification (0.0-1.0)

domain_confidence(Request, Domain, Confidence) :-
    domain_keywords(Domain, Keywords),
    keyword_count(Request, Keywords, Count),
    length(Keywords, TotalKeywords),

    % Confidence = (matches / total keywords) capped at 1.0
    Ratio is Count / TotalKeywords,
    min(Ratio, 1.0, Confidence).

% Helper: min/3
min(X, Y, X) :- X =< Y, !.
min(_, Y, Y).
```

---

5. Stakes Assessment Logic

5.1 Stakes Levels

```
stakes_level(low).
stakes_level(medium).
stakes_level(high).
```

5.2 Complexity Indicators

```
%% complexity_indicator(?Keyword, ?Score)
% Keywords indicating complexity (higher score = more complex)

% High complexity (score 3)
complexity_indicator(refactor, 3).
complexity_indicator(architecture, 3).
complexity_indicator(scalable, 3).
complexity_indicator('multi-file', 3).
complexity_indicator(distributed, 3).
complexity_indicator(concurrent, 3).
complexity_indicator('real-time', 3).

% Medium complexity (score 2)
complexity_indicator(optimize, 2).
complexity_indicator(algorithm, 2).
complexity_indicator(performance, 2).
complexity_indicator(integrate, 2).
complexity_indicator(migrate, 2).

% Low complexity (score 1)
complexity_indicator(simple, 1).
complexity_indicator(basic, 1).
complexity_indicator(quick, 1).
complexity_indicator(straightforward, 1).
```

5.3 Risk Indicators

```
%% risk_indicator(?Keyword, ?Score)
% Keywords indicating risk/criticality (higher = more risky)

% High risk (score 3)
risk_indicator(production, 3).
risk_indicator(critical, 3).
risk_indicator(security, 3).
risk_indicator(payment, 3).
risk_indicator(financial, 3).
risk_indicator(medical, 3).
risk_indicator('safety-critical', 3).

% Medium risk (score 2)
risk_indicator(important, 2).
risk_indicator(customer-facing, 2).
risk_indicator(api, 2).
risk_indicator(database, 2).

% Low risk (score 1)
risk_indicator(prototype, 1).
risk_indicator(experiment, 1).
risk_indicator(draft, 1).
risk_indicator(internal, 1).
```

5.4 Stakes Assessment Rules

```
%% assess_stakes(+Request, +Domain, -Stakes)
% Determine stakes level based on complexity and risk indicators

assess_stakes(Request, Domain, Stakes) :-
    % Calculate complexity score
    findall(Score,
            (complexity_indicator(Keyword, Score),
             sub_atom(Request, _, _, _, Keyword)),
            ComplexityScores),
    sum_list(ComplexityScores, ComplexityTotal),

    % Calculate risk score
    findall(Score,
            (risk_indicator(Keyword, Score),
             sub_atom(Request, _, _, _, Keyword)),
            RiskScores),
    sum_list(RiskScores, RiskTotal),

    % Combined score
    CombinedScore is ComplexityTotal + RiskTotal,

    % Determine stakes
    stakes_from_score(CombinedScore, Domain, Stakes).

%% stakes_from_score(+Score, +Domain, -Stakes)
% Map numeric score to stakes level

stakes_from_score(Score, _, high) :-
    Score >= 5,  % High complexity + high risk
    !.

stakes_from_score(Score, _, medium) :-
    Score >= 2,
    !.

stakes_from_score(_, _, low) :-
    !.

% Domain-specific adjustments
stakes_from_score(Score, coding_architecture, high) :-
    Score >= 3,  % Architecture is inherently high-stakes
    !.
```

6. Model Selection Rules

6.1 Model Capabilities

```
%% model_capability(?Model, ?Domain, ?Proficiency)
% Define which models excel at which domains
% Proficiency: high (0.9), medium (0.7), low (0.5)

% Qwen Coder - Best for architecture
model_capability(qwen_coder_32b, coding_architecture, 0.95).
model_capability(qwen_coder_32b, coding_implementation, 0.85).
model_capability(qwen_coder_32b, reasoning, 0.70).
model_capability(qwen_coder_32b, documentation, 0.75).

% Nemotron - Best for implementation
model_capability(nemotron_30b, coding_implementation, 0.95).
model_capability(nemotron_30b, coding_architecture, 0.75).
model_capability(nemotron_30b, reasoning, 0.65).

% GPT-OSS - Balanced reasoning
model_capability(gpt_oss_20b, reasoning, 0.90).
model_capability(gpt_oss_20b, coding_architecture, 0.75).
model_capability(gpt_oss_20b, coding_implementation, 0.70).
model_capability(gpt_oss_20b, documentation, 0.80).

% MythoMax - Creative specialist
model_capability(mythomax_13b, creative, 0.95).
model_capability(mythomax_13b, documentation, 0.70).
model_capability(mythomax_13b, reasoning, 0.60).
```

6.2 Model Selection Logic

```prolog
%% select_model(+Domain, +Stakes, -Model, -Confidence)
% Select best model for domain and stakes

select_model(Domain, Stakes, Model, Confidence) :-
    % Find all models capable of this domain
    findall(
        (M, Proficiency),
        model_capability(M, Domain, Proficiency),
        Candidates
    ),

    % Sort by proficiency (descending)
    sort(2, @>=, Candidates, Sorted),

    % Pick best candidate
    Sorted = [(BestModel, Proficiency)|_],

    % Adjust confidence by stakes
    stakes_confidence_multiplier(Stakes, Multiplier),
    Confidence is Proficiency * Multiplier,

    Model = BestModel.

% Confidence multipliers based on stakes
stakes_confidence_multiplier(high, 0.9).   % High stakes: slightly lower confidence
(more cautious)
stakes_confidence_multiplier(medium, 1.0). % No adjustment
stakes_confidence_multiplier(low, 1.1).    % Low stakes: boost confidence

% Fallback if no model found
select_model(_, _, gpt_oss_20b, 0.5) :-
    !.  % Default to balanced GPT-OSS with low confidence
```

6.3 Model Specialization Rules

```prolog
%% Specialized rules for specific scenarios

% Architecture refactoring: always Qwen
select_model(coding_architecture, high, qwen_coder_32b, 0.95) :-
    !.  % Override general rule

% Performance-critical implementation: prefer Nemotron
select_model(coding_implementation, _, nemotron_30b, 0.92) :-
    % Check if request mentions performance
    % (This would be passed as additional context)
    !.

% Creative content: always MythoMax
select_model(creative, _, mythomax_13b, 0.95) :-
    !.
```

7. Validation Policy Selection

7.1 Policy Types

```
validation_policy_type(none).          % No validation (low stakes, user override)
validation_policy_type(end_stage).     % Validate after full generation
validation_policy_type(block_by_block). % Line-by-line proof-checking
```

7.2 Policy Selection Rules

```
%% validation_policy(+Stakes, -Policy)
% Determine validation policy based on stakes

validation_policy(high, block_by_block) :-
    !.  % High stakes: rigorous validation

validation_policy(medium, end_stage) :-
    !.  % Medium stakes: validate after generation

validation_policy(low, none) :-
    !.  % Low stakes: optional validation

% User override (passed as context)
validation_policy(_, UserPolicy) :-
    % Check if user explicitly requested a policy
    % (Handled by Python wrapper, passed as arg)
    nonvar(UserPolicy),
    !.
```

7.3 Validation Granularity

```
%% validation_granularity(+Policy, -Granularity)
% How finely to validate

validation_granularity(block_by_block, per_line).
validation_granularity(block_by_block, per_function).  % Alternative
validation_granularity(end_stage, per_output).
validation_granularity(none, no_validation).
```

---

8. Tool Requirement Detection

8.1 Tool Types

```
tool_type(ocr).
tool_type(vision).
tool_type(embeddings).
```

8.2 Tool Detection Rules

```
%% detect_tools(+Request, -ToolsList)
% Detect which multimodal tools are required

detect_tools(Request, Tools) :-
    findall(Tool, requires_tool(Request, Tool), Tools).

%% requires_tool(+Request, -Tool)
% Individual tool requirement rules

% OCR required if keywords indicate document processing
requires_tool(Request, ocr) :-
    contains_keywords(Request, [scan, document, pdf, image, ocr, extract, page]).

% Vision required if keywords indicate image analysis
requires_tool(Request, vision) :-
    contains_keywords(Request, [image, picture, photo, diagram, chart, figure, visual]
).

% Embeddings required for semantic search/retrieval
requires_tool(Request, embeddings) :-
    contains_keywords(Request, [similar, related, find, search, retrieve, context, pas
t]).

% Embeddings also used as fallback for ambiguous requests
requires_tool(Request, embeddings) :-
    classify_domain(Request, unknown),  % Unknown domain
    !.
```

---

9. Confidence Scoring

9.1 Overall Confidence Calculation

```
%% overall_confidence(+Domain, +Stakes, +Model, +ModelConfidence, -OverallConfidence)
% Calculate overall routing confidence

overall_confidence(Domain, Stakes, Model, ModelConfidence, Overall) :-
    % Factor 1: Domain classification confidence
    domain_confidence(Domain, DomainConf),

    % Factor 2: Model selection confidence (passed in)
    % ModelConfidence already calculated

    % Factor 3: Stakes assessment certainty
    stakes_certainty(Stakes, StakesConf),

    % Combined: weighted average
    Overall is (DomainConf * 0.4 + ModelConfidence * 0.4 + StakesConf * 0.2).

%% stakes_certainty(+Stakes, -Confidence)
% How confident are we in the stakes assessment?

stakes_certainty(high, 0.9).    % Clear indicators
stakes_certainty(medium, 0.8).  % Some indicators
stakes_certainty(low, 0.95).    % Default (absence of indicators is certain)
```

9.2 Confidence Thresholds

```
%% confidence_threshold(?Level, ?Threshold)
% Thresholds for different confidence actions

confidence_threshold(acceptable, 0.75).  % Proceed normally
confidence_threshold(uncertain, 0.60).   % Trigger fallback (embeddings)
confidence_threshold(critical, 0.50).    % Require user clarification
```

---

10. Uncertainty Handling

10.1 Fallback Mechanisms

```
%% handle_uncertainty(+Request, +InitialDecision, -FinalDecision)
% Handle low-confidence routing decisions

handle_uncertainty(Request, InitialDecision, FinalDecision) :-
    InitialDecision.confidence < 0.75,  % Below acceptable threshold

    % Trigger embedding-based fallback
    % (Python will call semantic similarity search)

    % Log uncertainty
    uncertainty_note(InitialDecision, Note),

    % Add uncertainty flag to decision
    FinalDecision = InitialDecision.put(_{
        uncertainty: true,
        uncertainty_note: Note,
        fallback_triggered: embeddings
    }).

handle_uncertainty(_, Decision, Decision).  % Pass through if confident

%% uncertainty_note(+Decision, -Note)
% Generate human-readable uncertainty explanation

uncertainty_note(Decision, Note) :-
    format(atom(Note),
           'Low confidence (~w) in domain classification (~w). Using embedding fall-
back.',
           [Decision.confidence, Decision.domain]).
```

10.2 Conflict Resolution

```
%% resolve_conflict(+Candidate1, +Candidate2, -Winner)
% If multiple rules match, resolve conflict

resolve_conflict(C1, C2, Winner) :-
    C1.confidence > C2.confidence,
    Winner = C1,
    !.

resolve_conflict(_, C2, C2).  % C2 wins if equal or higher confidence
```

11. Learning & Adaptation

11.1 Feedback Integration

```
%% record_feedback(+RequestID, +ActualDomain, +ActualModel, +Outcome)
% Record user feedback or validation outcomes to improve routing

record_feedback(RequestID, ActualDomain, ActualModel, Outcome) :-
    % Store in knowledge_graph.md via Python
    % Future: Dynamically adjust routing rules based on patterns

    % Example: If Qwen frequently fails on domain X, reduce its confidence
    % This is a placeholder for future learning logic
    true.
```

11.2 Dynamic Rule Adjustment

```
%% adjust_model_confidence(+Model, +Domain, +Adjustment)
% Dynamically adjust model proficiency scores based on performance

adjust_model_confidence(Model, Domain, Adjustment) :-
    % Retract old capability fact
    retract(model_capability(Model, Domain, OldProf)),

    % Calculate new proficiency
    NewProf is max(0.1, min(1.0, OldProf + Adjustment)),

    % Assert new capability
    assert(model_capability(Model, Domain, NewProf)).
```

12. Python-Prolog Integration

12.1 Python Wrapper

```python
from pyswip import Prolog
import json
from typing import Dict, Any

class PrologRouter:
    """
    Python wrapper for Prolog routing logic.
    """

    def __init__(self, prolog_file: str = "config/routing_rules.pl"):
        self.prolog = Prolog()
        self.prolog.consult(prolog_file)

    def classify_request(self, user_input: str, context: Dict = None) -> Dict[str,
Any]:
        """
        Classify user request using Prolog rules.

        Args:
            user_input: User's task description
            context: Additional context (history, preferences, etc.)

        Returns:
            Routing decision as dict
        """
        # Escape quotes in input
        escaped_input = user_input.replace("'", "\\'")

        # Query Prolog
        query = f"route('{escaped_input}', Decision)"

        try:
            results = list(self.prolog.query(query))

            if results:
                decision_dict = results[0]['Decision']

                # Convert Prolog dict to Python dict
                return self._prolog_dict_to_python(decision_dict)
            else:
                # Fallback if no match
                return self._fallback_decision()

        except Exception as e:
            print(f"Prolog query error: {e}")
            return self._fallback_decision()

    def _prolog_dict_to_python(self, prolog_dict) -> Dict[str, Any]:
        """Convert Prolog dict to Python dict."""
        # pyswip returns dict-like objects; convert to standard dict
        return {
            "domain": str(prolog_dict.get('domain', 'unknown')),
            "stakes": str(prolog_dict.get('stakes', 'medium')),
            "recommended_model": str(prolog_dict.get('recommended_model', 'gpt_oss_20b
')),
            "validation_policy": str(prolog_dict.get('validation_policy', 'end_stage')
),
            "tools_required": prolog_dict.get('tools_required', []),
            "confidence": float(prolog_dict.get('confidence', 0.5)),
            "reasoning": str(prolog_dict.get('reasoning', 'Default routing'))
        }
```

```python
    def _fallback_decision(self) -> Dict[str, Any]:
        """Fallback decision if Prolog fails."""
        return {
            "domain": "unknown",
            "stakes": "medium",
            "recommended_model": "gpt_oss_20b",
            "validation_policy": "end_stage",
            "tools_required": ["embeddings"],
            "confidence": 0.4,
            "reasoning": "Prolog routing failed; using safe default"
        }
```

12.2 Integration with Router Model

```python
class HybridRouter:
    """
    Combines neural Router Model with Prolog rules.
    """

    def __init__(self):
        self.router_model = RouterModel()  # Granite-Micro 3B
        self.prolog_router = PrologRouter()
        self.embeddings = EmbeddingService()

    def route(self, user_input: str) -> Dict[str, Any]:
        """
        Hybrid routing: Neural model + Prolog refinement + Embedding fallback.
        """
        # Step 1: Neural model (fast, initial classification)
        neural_output = self.router_model.classify(user_input)

        # Step 2: Prolog refinement (apply logical rules)
        prolog_output = self.prolog_router.classify_request(user_input)

        # Step 3: Merge decisions (Prolog takes precedence if confident)
        if prolog_output['confidence'] >= 0.75:
            final_decision = prolog_output
        elif neural_output['confidence'] >= 0.80:
            final_decision = neural_output
        else:
            # Step 4: Embedding fallback (low confidence)
            similar_tasks = self.embeddings.find_similar(user_input, top_k=3)
            final_decision = self._infer_from_similar(similar_tasks)
            final_decision['fallback'] = 'embeddings'

        return final_decision
```

---

13. Testing & Validation

13.1 Test Dataset

Format: CSV or JSON

request	expected_domain	expected_stakes	expected_model	notes
"Refactor payment module for SOLID principles"	coding_architecture	high	qwen_coder_32b	Architecture focus, high stakes
"Write a function to sort a list in Python"	coding_implementation	low	nemotron_30b	Simple implementation
"Analyze pros/cons of microservices vs monolith"	reasoning	medium	gpt_oss_20b	Balanced reasoning task
"Write a blog post about AI ethics"	creative	low	mythomax_13b	Creative content

13.2 Accuracy Metrics

```
def evaluate_routing_accuracy(test_dataset, router):
    """
    Evaluate routing accuracy on labeled test dataset.
    """
    correct_domain = 0
    correct_stakes = 0
    correct_model = 0
    total = len(test_dataset)

    for test_case in test_dataset:
        decision = router.route(test_case['request'])

        if decision['domain'] == test_case['expected_domain']:
            correct_domain += 1

        if decision['stakes'] == test_case['expected_stakes']:
            correct_stakes += 1

        if decision['recommended_model'] == test_case['expected_model']:
            correct_model += 1

    return {
        'domain_accuracy': correct_domain / total,
        'stakes_accuracy': correct_stakes / total,
        'model_accuracy': correct_model / total,
        'overall_accuracy': (correct_domain + correct_stakes + correct_model) / (total
* 3)
    }
```

Target Accuracy:
- Domain classification: ≥90%

- Stakes assessment: ≥85%
- Model selection: ≥90%

13.3 Unit Tests (Prolog)

```
%% test_domain_classification/0
% Unit test for domain classification

test_domain_classification :-
    classify_domain("Refactor architecture for scalability", coding_architecture),
    classify_domain("Write a function to calculate fibonacci", coding_implementation),
    classify_domain("Compare SQL vs NoSQL databases", reasoning),
    classify_domain("Write a poem about AI", creative),
    !.  % All tests must pass

%% run_all_tests/0
run_all_tests :-
    test_domain_classification,
    write('All tests passed!'), nl.
```

---

14. Appendices

14.1 Complete Routing Rules File

File: `config/routing_rules.pl`

```prolog
% ============================================================================
% Sovereign AI Infrastructure - Routing Rules
% File: routing_rules.pl
% Version: 1.0
% Date: 2026-02-05
% ============================================================================

% Load utility modules
:- use_module(library(lists)).
:- use_module(library(aggregate)).

% ============================================================================
% 1. FACTS & CONSTANTS
% ============================================================================

% Domain types
domain_type(coding_architecture).
domain_type(coding_implementation).
domain_type(reasoning).
domain_type(creative).
domain_type(documentation).
domain_type(unknown).

% Stakes levels
stakes_level(low).
stakes_level(medium).
stakes_level(high).

% Validation policies
validation_policy_type(none).
validation_policy_type(end_stage).
validation_policy_type(block_by_block).

% Tool types
tool_type(ocr).
tool_type(vision).
tool_type(embeddings).

% ============================================================================
% 2. DOMAIN KEYWORDS
% ============================================================================

domain_keywords(coding_architecture, [
    refactor, architecture, design, pattern, solid, dependency, interface,
    abstraction, coupling, cohesion, 'design pattern', microservice,
    'system design', modular, scalable, extensible
]).

domain_keywords(coding_implementation, [
    implement, 'write function', 'write class', create, build, develop,
    optimize, performance, 'bug fix', debug, test, algorithm, script
]).

domain_keywords(reasoning, [
    analyze, evaluate, assess, plan, strategy, decide, compare, explain,
    reason, logic, problem, solution, approach, tradeoff, recommendation
]).

domain_keywords(creative, [
    write, story, narrative, blog, article, essay, creative, poem,
    draft, tone, style, engaging, marketing, copy, tweet
]).
```

```
domain_keywords(documentation, [
    document, readme, manual, guide, tutorial, explain, describe,
    'how to', overview, summary, report, specification
]).

% ============================================================================
% 3. COMPLEXITY & RISK INDICATORS
% ============================================================================

complexity_indicator(refactor, 3).
complexity_indicator(architecture, 3).
complexity_indicator('multi-file', 3).
complexity_indicator(optimize, 2).
complexity_indicator(algorithm, 2).
complexity_indicator(simple, 1).
complexity_indicator(basic, 1).

risk_indicator(production, 3).
risk_indicator(critical, 3).
risk_indicator(security, 3).
risk_indicator(payment, 3).
risk_indicator(important, 2).
risk_indicator(prototype, 1).

% ============================================================================
% 4. MODEL CAPABILITIES
% ============================================================================

model_capability(qwen_coder_32b, coding_architecture, 0.95).
model_capability(qwen_coder_32b, coding_implementation, 0.85).
model_capability(nemotron_30b, coding_implementation, 0.95).
model_capability(nemotron_30b, coding_architecture, 0.75).
model_capability(gpt_oss_20b, reasoning, 0.90).
model_capability(gpt_oss_20b, coding_architecture, 0.75).
model_capability(mythomax_13b, creative, 0.95).

% ============================================================================
% 5. CORE ROUTING LOGIC (see sections 3-10 above)
% ============================================================================

% [Insert all predicates from sections 3-10]

% ============================================================================
% END OF ROUTING RULES
% ============================================================================
```

14.2 Example Queries

```
% Example 1: Simple routing
?- route("Refactor payment module for SOLID principles", Decision).

% Example 2: Check domain classification
?- classify_domain("Write a function to sort a list", Domain).

% Example 3: Assess stakes
?- assess_stakes("Critical production bug in payment system", coding_implementation, Stakes).

% Example 4: Select model
?- select_model(coding_architecture, high, Model, Confidence).

% Example 5: Detect tools
?- detect_tools("Extract text from this scanned medical record", Tools).
```

Document Status: Draft for Review
Next Steps: Review by Technical Lead, ML Lead, Backend Lead; Implementation and testing
Target Approval Date: 2026-02-12
Owner: Logic Systems Engineer / Routing Lead

End of Routing Logic Specification Document