Contents

RADIANT Orchestration Patterns System

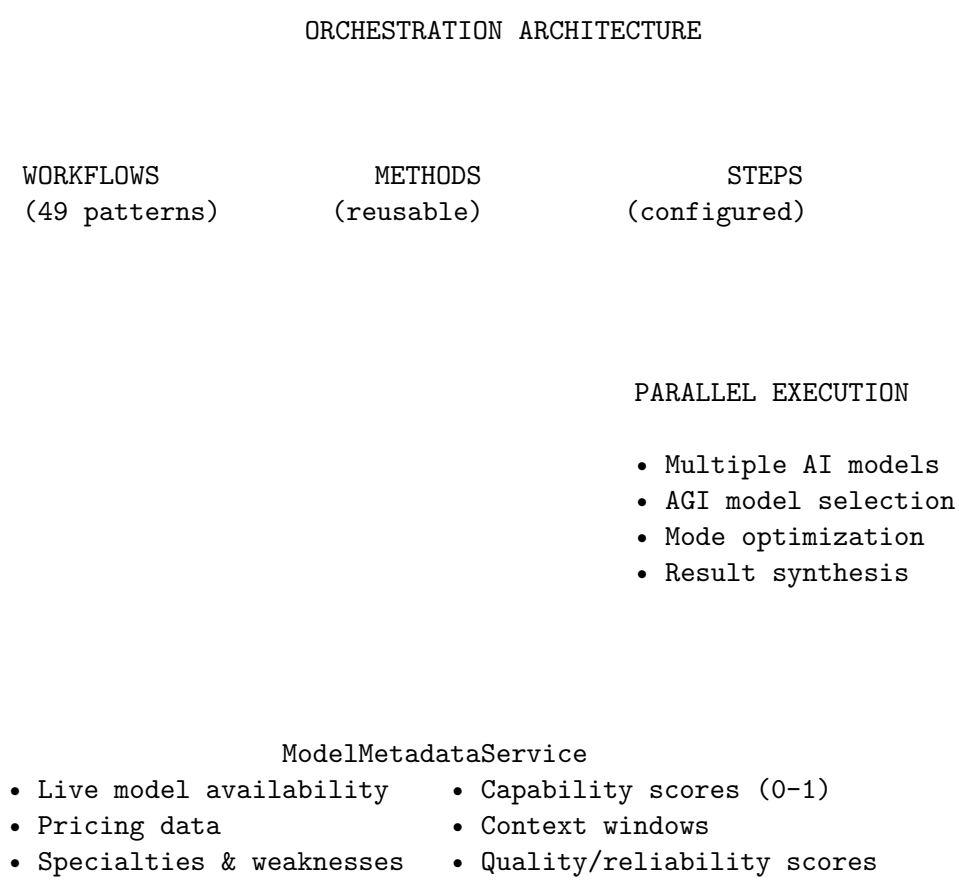
Version: 4.18.0
Last Updated: December 2024

Overview

The RADIANT Orchestration Patterns System enables sophisticated multi-AI workflows that leverage multiple AI providers in parallel, with intelligent model selection based on task characteristics and domain analysis.

Table of Contents

Architecture



Key Components

Component	Location	Purpose
OrchestrationPatternsService	`packages/infrastructure/lambda/shared/services/orchestration-pat`	Catalog and manage workflow execution
ModelMetadataService	`packages/infrastructure/lambda/shared/services/model-metadata.se`	Track model capabilities
Visual Editor	`apps/admin-dashboard/app/(dashboard)/orchestration-patterns/edit`	Admin UI for workflow design
Shared Components	`apps/admin-dashboard/components/workflow-editor/index.tsx`	Reusable UI components

Orchestration Workflows

49 Documented Patterns

Workflows are organized into categories:

Category	Patterns	Example
Consensus & Aggregation	Self-Consistency, Universal Self-Consistency, Meta-Reasoning	Multiple samples with majority voting
Debate & Deliberation	AI Debate, Multi-Agent Debate, Cross-Examination	Adversarial argumentation
Critique & Refinement	Self-Refine, Reflexion, Constitutional AI	Iterative improvement
Verification & Validation	Chain-of-Verification, Fact-Checking Pipeline	Multi-stage fact checking
Decomposition	Least-to-Most, Decomposed Prompting, Tree of Thoughts	Problem breakdown
Specialized Reasoning	Chain-of-Thought, ReAct, Graph-of-Thoughts	Enhanced reasoning patterns
Multi-Model Routing	Mixture of Experts, Speculative Decoding, Model Cascading	Intelligent routing
Ensemble Methods	Model Ensemble, Boosted Prompting, Blended RAG	Multiple model combination

Workflow Structure

```typescript
interface OrchestrationWorkflow {
  workflowId: string;
  workflowCode: string;          // e.g., "SOD" for AI Debate
  commonName: string;            // e.g., "AI Debate"
  formalName: string;            // e.g., "Scalable Oversight via Debate"
  category: string;
```

```
  categoryCode: string;
  patternNumber: number;          // 1-49
  description: string;
  detailedDescription?: string;
  bestFor: string[];              // Use cases
  problemIndicators: string[];    // When to use
  qualityImprovement: string;     // Expected improvement
  typicalLatency: string;
  typicalCost: string;
  minModelsRequired: number;
  defaultConfig: Record<string, unknown>;
  isSystemWorkflow: boolean;
  isEnabled: boolean;
}
```

Methods & Steps

Reusable Methods

Methods are shared building blocks with default parameters:

Method Code	Name	Role	Description
`GENERATE_RESPONSE`	Generate Response	generator	Generate a response using AI model
`GENERATE_WITH_COT`	Chain-of-Thought	generator	Generate with step-by-step reasoning
`CRITIQUE_RESPONSE`	Critique Response	critic	Critically evaluate for flaws
`JUDGE_RESPONSES`	Judge Responses	judge	Compare and judge multiple responses
`VERIFY_FACTS`	Verify Facts	verifier	Extract and verify factual claims
`SYNTHESIZE_RESPONSES`	Synthesize	synthesizer	Combine best parts from multiple
`BUILD_CONSENSUS`	Build Consensus	synthesizer	Identify points of agreement
`GENERATE_CHALLENGE`	Challenge	challenger	Argue opposite position
`DEFEND_POSITION`	Defend	defender	Defend against challenges
`DECOMPOSE_PROBLEM`	Decompose	reasoner	Break down complex problems
`MAJORITY_VOTE`	Majority Vote	aggregator	Select most common answer
`WEIGHTED_AGGREGATE`	Weighted Aggregate	aggregator	Combine weighted by confidence

Workflow Steps

Steps are method instances with custom configuration:

```typescript
interface WorkflowStep {
  bindingId: string;
  stepOrder: number;
  stepName: string;
  stepDescription?: string;
  method: OrchestrationMethod;
  parameterOverrides: Record<string, unknown>;  // Override defaults
  conditionExpression?: string;                 // Conditional execution
  isIterative: boolean;                         // Repeat execution
  maxIterations: number;
  iterationCondition?: string;
  dependsOnSteps: number[];                     // DAG dependencies
  modelOverride?: string;                       // Force specific model
  outputVariable?: string;                      // Store output
  parallelExecution?: ParallelExecutionConfig;  // Parallel AI calls
}
```

Parallel Execution

Each method step can call **multiple AI providers simultaneously** for improved quality and reliability.

Execution Modes

Mode	Behavior	Best For
all	Wait for all models to respond	Maximum quality, comprehensive synthesis
race	Return first successful response	Latency-sensitive applications
quorum	Continue when X% of models respond	Balance of speed and quality

Synthesis Strategies

Strategy	How It Works
best_of	Select response with highest confidence score
vote	Choose most common answer pattern (majority vote)
weighted	Score by confidence × speed, select highest
merge	Combine insights from all models into unified response

Configuration

```typescript
interface ParallelExecutionConfig {
  enabled: boolean;
  mode: 'all' | 'race' | 'quorum';
  models: string[];                    // Fallback if AGI disabled
```

```
  quorumThreshold?: number;            // 0.5 = majority
  synthesizeResults?: boolean;
  synthesisStrategy?: 'best_of' | 'merge' | 'vote' | 'weighted';
  weightByConfidence?: boolean;
  timeoutMs?: number;                  // Per-model timeout
  failureStrategy?: 'fail_fast' | 'continue' | 'fallback';

  // AGI Dynamic Selection
  agiModelSelection?: boolean;         // Enable AGI selection
  minModels?: number;                  // Min models to select (default: 2)
  maxModels?: number;                  // Max models to select (default: 5)
  domainHints?: string[];              // Hints for domain detection
  preferredModes?: ModelMode[];        // Preferred execution modes
}
```

AGI Dynamic Model Selection

When `agiModelSelection` is enabled, the system **dynamically selects optimal models** based on:

1. Domain Detection

Analyzes prompt content to detect subject domain:

Domain	Keywords Detected
coding	code, function, class, debug, algorithm, typescript, python, api, database
math	calculate, equation, formula, proof, theorem, algebra, calculus, integral
science	scientific, hypothesis, experiment, physics, chemistry, biology, quantum
legal	legal, contract, law, regulation, compliance, liability, jurisdiction
medical	medical, diagnosis, treatment, symptoms, patient, clinical, therapy
finance	financial, investment, market, stock, trading, portfolio, valuation
creative	write, story, poem, creative, narrative, fiction, imagine, design
reasoning	reason, logic, deduce, infer, conclude, argue, step by step
research	research, comprehensive, thorough, deep dive, explore, investigate

2. Task Characteristics

Analyzes prompt for task requirements:

```
interface TaskCharacteristics {
  complexity: 'low' | 'medium' | 'high';
  requiresReasoning: boolean;     // "think", "step by step", "why"
  requiresCreativity: boolean;    // "creative", "imagine", "write"
  requiresPrecision: boolean;     // "exact", "precise", "accurate"
```

```
  requiresResearch: boolean;      // "research", "comprehensive", "thorough"
  estimatedTokens: number;
}
```

3. Live Model Scoring

Queries `ModelMetadataService` for available models and scores based on:

- **Domain match** from model specialties
- **Capability scores** (reasoning, coding, creative, etc.)
- **Quality/reliability scores** from metadata
- **Context window** for complex tasks
- **Mode compatibility** for task type
- **Cost efficiency** for budget-conscious selection

4. Optimal Mode Assignment

For each selected model, assigns the optimal execution mode.

Model Modes

Modes configure how models are invoked based on their capabilities and task requirements.

Available Modes

Mode	Icon	Description	Auto-Selected When	Parameters Applied
standard	-	Default execution	Fallback	Default params
thinking		Extended reasoning	`requiresReasoning=true` + o1/claude/r1	`thinkingBudget: 5000-10000, enableThinking: true`
deep_research		In-depth research	`requiresResearch=true` + perplexity/gemini-deep	`searchDepth: comprehensive, includeSources: true`
fast		Speed-optimized	flash/turbo/mini models	`maxTokens: 2048, streamResponse: true`
creative		Higher temperature	`requiresCreativity=true`	`temperature: 0.9, topP: 0.95`
precise		Low temperature	`requiresPrecision=true`	`temperature: 0.1, topP: 0.9`
code		Code-specialized	coding domain	`temperature: 0.2`
vision		Multimodal vision	vision-capable models	`enableVision: true`
long_context		Extended context	large context windows	`maxTokens: 16384, useLongContext: true`

Mode Selection Logic

```
// Example: Thinking mode selection
if (characteristics.requiresReasoning) {
```

```
  if (modelId.includes('o1') || modelId.includes('o3')) {
    return { mode: 'thinking', modeBonus: 0.3 };
  }
  if (modelId.includes('claude') && modelId.includes('3.5')) {
    return { mode: 'thinking', modeBonus: 0.25 };
  }
  if (modelId.includes('deepseek') && modelId.includes('r1')) {
    return { mode: 'thinking', modeBonus: 0.25 };
  }
}

// Example: Deep research mode selection
if (characteristics.requiresResearch) {
  if (modelId.includes('perplexity') || modelId.includes('sonar')) {
    return { mode: 'deep_research', modeBonus: 0.35 };
  }
  if (modelId.includes('gemini') && modelName.includes('deep')) {
    return { mode: 'deep_research', modeBonus: 0.3 };
  }
}
```

Visual Workflow Editor

Features

- **Method Palette** - Drag-and-drop orchestration methods
- **Canvas** - Visual workflow design with nodes and connections
- **Step Configuration** - 4-tab panel:
  - **General** - Name, order, model override, output variable
  - **Parameters** - JSON overrides with quick editors
  - **Parallel** - AGI selection, modes, synthesis
  - **Advanced** - Iteration, conditions
- **Zoom/Pan** - Canvas navigation
- **Settings Dialog** - Workflow-level configuration

Parallel Tab Configuration

```
Enable Parallel Execution                    [ ]

AGI Model Selection                          [ ]

Min Models: [2]      Max Models: [5]
Domain Hints: coding, reasoning
Preferred Modes:
  [ ] thinking  [ ] deep_research  [ ] fast
  [ ] creative  [ ] precise        [ ] code
```

```
  Execution Mode: [All (wait for all models)     ]
  Quorum Threshold: [    ] 50%

  [ ] Synthesize Results
  Strategy: [Weighted (confidence + speed)       ]

  Timeout: [30000] ms
  Failure Strategy: [Continue (use successful)   ]
```

API Reference

OrchestrationPatternsService

```typescript
class OrchestrationPatternsService {
  // Pattern Selection
  async selectPattern(request: PatternSelectionRequest): Promise<PatternSelectionResult>;

  // Workflow Execution
  async executeWorkflow(request: ExecutionRequest): Promise<ExecutionResult>;

  // CRUD Operations
  async getWorkflow(workflowCode: string): Promise<OrchestrationWorkflow | null>;
  async getWorkflowSteps(workflowId: string): Promise<WorkflowStep[]>;
  async getAllWorkflows(options?: { category?: string; enabledOnly?: boolean }): Promise<Orches
  async getMethods(category?: string): Promise<OrchestrationMethod[]>;
}
```

Execution Flow

```typescript
// 1. Select best pattern for task
const selection = await orchestrationPatternsService.selectPattern({
  tenantId: 'tenant-123',
  prompt: 'Write a recursive algorithm for TSP with dynamic programming',
  taskType: 'coding',
  complexity: 'high',
  qualityPriority: 0.9,
});

// 2. Execute selected workflow
const result = await orchestrationPatternsService.executeWorkflow({
  tenantId: 'tenant-123',
  workflowCode: selection.selectedPattern.workflowCode,
  prompt: '...',
  configOverrides: {
```

```
    parallelExecution: {
      enabled: true,
      agiModelSelection: true,
      minModels: 3,
      preferredModes: ['thinking', 'code'],
    },
  },
});

// 3. Result includes all step outputs
console.log(result.response);           // Final synthesized response
console.log(result.qualityScore);       // 0-1 quality assessment
console.log(result.steps);              // Individual step results
console.log(result.modelsUsed);         // All models that participated
```

Step Execution Result

```
interface StepExecutionResult {
  stepOrder: number;
  stepName: string;
  methodCode: string;
  input: Record<string, unknown>;
  output: Record<string, unknown>;
  modelUsed: string;                    // Primary model
  latencyMs: number;
  costCents: number;
  iteration: number;

  // Parallel execution details
  wasParallel?: boolean;
  parallelResults?: ParallelExecutionResult[];
  synthesizedFrom?: string[];           // Models that contributed
}

interface ParallelExecutionResult {
  modelId: string;
  response: string;
  latencyMs: number;
  costCents: number;
  tokensUsed: number;
  confidence?: number;                  // 0-1 estimated confidence
  status: 'success' | 'failed' | 'timeout';
  error?: string;
}
```

Database Schema

Key tables in `packages/infrastructure/migrations/066_orchestration_patterns_registry.sql`:

```
-- Core tables
orchestration_methods          -- Reusable method definitions
orchestration_workflows        -- 49 workflow patterns
workflow_method_bindings       -- Steps linking workflows to methods
workflow_customizations        -- Per-tenant/user overrides

-- Execution tracking
orchestration_executions       -- Workflow execution records
orchestration_step_executions  -- Individual step records
```

Best Practices

1. When to Enable AGI Selection

Enable when: - Task domain is unclear or mixed - Maximum quality is required - Cost is not a primary concern

Disable when: - Specific model is required (compliance) - Predictable cost is critical - Testing specific model behavior

2. Choosing Execution Mode

Use Case	Recommended Mode
Critical decisions	`all` with `vote` synthesis
User-facing latency-sensitive	`race`
Background processing	`all` with `merge` synthesis
Cost-sensitive	`quorum` at 50%

3. Mode Selection Tips

- Enable **thinking** mode for math, reasoning, complex analysis
- Enable **deep_research** mode for fact-finding, comprehensive answers
- Enable **fast** mode for simple queries, autocomplete
- Enable **code** mode for programming tasks
- Enable **precise** mode for factual, accuracy-critical responses

---

Troubleshooting

Common Issues

Models not being selected: - Check `ModelMetadataService` has available models - Verify `isAvailable: true` in model metadata - Check domain hints match model specialties

High latency: - Reduce `maxModels` - Use `race` mode instead of `all` - Disable `thinking` mode for simple tasks

Inconsistent results: - Enable `synthesizeResults` with `vote` strategy - Increase `minModels` for more consensus - Use `precise` mode for factual tasks

---

Changelog

v4.18.0 (December 2024)

- Added dynamic model selection from `ModelMetadataService`
- Added 9 model execution modes (thinking, deep_research, etc.)
- Added AGI-driven mode assignment based on task analysis
- Removed hardcoded model lists
- Added preferred modes configuration in UI
- Enhanced domain detection with research category
- Added mode-specific parameter application