Cross-Domain Solution Matcher: A Multi-Stage DSPy Pipeline for Analogical Problem Solving

Architecture and Implementation

Tyler Gibbs
Cofounder, AI & Engineering
Grayhaven

September 2025

Table of Contents

Abstract

This paper presents the Cross-Domain Solution Matcher (CDSM), a novel multi-stage DSPy pipeline that leverages large language models to find analogous solutions from disparate domains. By abstracting problems to their core challenges, mapping them to diverse fields, extracting domain-specific solutions, and synthesizing actionable recommendations, CDSM enables creative problem-solving through cross-domain analogical reasoning. The system demonstrates how structured prompting frameworks can orchestrate complex reasoning tasks while maintaining interpretability and modularity.

TL;DR: CDSM is a 4-stage pipeline (analyze → map → extract → synthesize) that solves problems by finding creative solutions from completely different domains—like using immune system strategies to reduce customer churn.

Introduction

The ability to transfer solutions across domains represents one of the most powerful forms of creative problem-solving. When biologists study ant colonies to optimize network routing, or when engineers apply fluid dynamics to understand crowd movement, they engage in analogical reasoning—recognizing structural similarities between superficially different domains.

The Cross-Domain Solution Matcher automates this process by decomposing analogical reasoning into discrete, optimizable stages. Rather than treating cross-domain thinking as a monolithic task, CDSM breaks it into four fundamental operations: problem abstraction, domain identification, solution extraction, and synthesis. Each stage builds upon the previous one, creating a pipeline where specialized reasoning modules collaborate to produce novel insights.

Reasoning from First Principles

> **Core Principle:** Problems that **look** different may share the same **underlying structure**.

At its core, analogical reasoning rests on a simple premise. Consider these two scenarios:

1. A glacier flowing down a mountain
2. A queue of customers waiting at a service counter

On the surface, these share nothing. Yet both involve:

- Flow constrained by capacity
- Buildup when input exceeds output
- Dynamics influenced by environmental factors

By stripping away surface details to reveal structural essence, we can transport solutions between domains. A fluid dynamics model for glaciers might inform queue management strategies. This is not metaphor—it's structural isomorphism.

CDSM operationalizes this principle through explicit abstraction: the first stage extracts the "core challenge" independent of domain-specific details. This abstraction then serves as a bridge, allowing the system to identify structurally similar problems in radically different fields.

Architecture Overview

The Cross-Domain Solution Matcher consists of four sequential modules, each implemented as a DSPy module using Chain-of-Thought reasoning. Table 1 summarizes the pipeline structure.

Stage	Input	Output	Purpose
Analysis	Problem description	Core challenge, characteristics	Abstract to essence
Mapping	Core challenge	5-7 analogous domains	Divergent search
Extraction	Domain + challenge	Solutions, strategies	Transfer knowledge
Synthesis	All solutions	Patterns, recommendations	Convergent integration

Table 1: Pipeline stage breakdown with inputs, outputs, and purposes

Stage 1: Problem Analyzer

Purpose: Extract the essence of the problem independent of its original domain.

Input: Raw problem description from the user

Output:

- Core challenge (the fundamental issue stripped of context)
- Key characteristics (structural properties of the problem)
- Desired outcome (what success looks like)

```python
class ProblemAnalyzer(dspy.Module):
    def __init__(self):
        super().__init__()
        self.analyze = dspy.ChainOfThought(AnalyzeProblem)

    def forward(self, problem_description):
        return self.analyze(
            problem_description=problem_description
        )
```

Listing 1: Problem Analyzer implementation

The Problem Analyzer serves as the abstraction engine. By forcing the model to articulate problems in domain-agnostic terms, we create a representation suitable for cross-domain mapping.

Stage 2: Domain Mapper

Purpose: Identify analogous domains where structurally similar problems have been solved.

Input: Core challenge and key characteristics from Stage 1

Output:

- List of 5-7 analogous domains
- Reasoning for each domain selection

```python
class DomainMapper(dspy.Module):
    def __init__(self):
        super().__init__()
        self.identify = dspy.ChainOfThought(IdentifyDomains)

    def forward(self, core_challenge, key_characteristics):
        return self.identify(
            core_challenge=core_challenge,
            key_characteristics=key_characteristics
        )
```

Listing 2: Domain Mapper implementation

The Domain Mapper performs divergent search across human knowledge. Given an abstract problem structure, it seeks domains where similar patterns appear—often in unexpected places. The diversity of domains selected directly influences solution creativity.

Stage 3: Solution Extractor

Purpose: For each identified domain, extract specific solutions and strategies.

Input: Original problem, core challenge, and target domain

Output (per domain):

- Domain-specific solution
- Mapping strategy (how the solution relates to the original problem)
- Implementation ideas (concrete steps to adapt the solution)

```python
class SolutionExtractor(dspy.Module):
    def __init__(self):
        super().__init__()
        self.extract = dspy.ChainOfThought(ExtractSolution)

    def forward(self, original_problem, core_challenge,
                target_domain):
        return self.extract(
            original_problem=original_problem,
            core_challenge=core_challenge,
            target_domain=target_domain
        )
```

Listing 3: Solution Extractor implementation

This stage performs the critical mapping operation—translating domain-specific knowledge back to the original problem context. The Solution Extractor must maintain both fidelity to domain expertise and relevance to the target problem.

Stage 4: Solution Synthesizer

Purpose: Integrate insights across all domains into coherent recommendations.

Input: Original problem and all domain solutions

Output:

- Common patterns across domains

- Unique insights from specific domains
- Prioritized, actionable recommendations

```python
class SolutionSynthesizer(dspy.Module):
    def __init__(self):
        super().__init__()
        self.synthesize = dspy.ChainOfThought(SynthesizeSolutions)

    def forward(self, original_problem, domain_solutions):
        return self.synthesize(
            original_problem=original_problem,
            domain_solutions=domain_solutions
        )
```

Listing 4: Solution Synthesizer implementation

The Synthesizer performs convergent integration, identifying meta-patterns that emerge across multiple domains while preserving domain-specific insights that might be uniquely valuable.

Pipeline Execution

The complete pipeline executes as shown in Listing 5:

```python
class CrossDomainSolutionMatcher(dspy.Module):
    def __init__(self, max_domains=5):
        super().__init__()
        self.max_domains = max_domains
        self.problem_analyzer = ProblemAnalyzer()
        self.domain_mapper = DomainMapper()
        self.solution_extractor = SolutionExtractor()
        self.solution_synthesizer = SolutionSynthesizer()

    def forward(self, problem_description):
        # Stage 1: Analyze the problem
        analysis = self.problem_analyzer(problem_description)

        # Stage 2: Map to analogous domains
        domain_mapping = self.domain_mapper(
            analysis.core_challenge,
            analysis.key_characteristics
        )

        # Stage 3: Extract solutions from each domain
        domain_solutions = []
        for domain in domain_mapping.analogous_domains[:self.max_domains]:
            solution = self.solution_extractor(
                problem_description,
                analysis.core_challenge,
                domain
            )
            domain_solutions.append(solution)

        # Stage 4: Synthesize all solutions
        synthesis = self.solution_synthesizer(
            problem_description,
            domain_solutions
        )

        return dspy.Prediction(
            problem=problem_description,
            core_challenge=analysis.core_challenge,
            domains=domain_mapping.analogous_domains,
            solutions=domain_solutions,
            synthesis=synthesis
        )
```

Listing 5: Complete pipeline implementation

Design Principles

Chain-of-Thought for All Modules

Every module uses `dspy.ChainOfThought` rather than basic `dspy.Predict`. This decision stems from empirical observation: analogical reasoning requires explicit intermediate steps. By forcing the model to articulate its reasoning, we achieve:

1. **Better Quality:** Step-by-step reasoning catches logical gaps
2. **Interpretability:** Users can inspect how analogies were formed
3. **Debuggability:** Failures can be traced to specific reasoning steps

From First Principles: Complex tasks benefit from decomposition. Chain-of-Thought operationalizes this decomposition at the prompting level.

Multi-Stage Pipeline Architecture

The four-stage design reflects the natural cognitive process of analogical reasoning:

1. **Understanding** what the problem really is (abstraction)
2. **Recalling** where similar problems exist (memory search)
3. **Retrieving** specific solutions (knowledge extraction)
4. **Adapting** solutions to the current context (transfer)

Each stage has a single, well-defined responsibility. This modularity enables:

- Independent optimization of each stage
- Easy replacement of individual components
- Clear debugging boundaries

TL;DR on Design: Each stage = one cognitive step; Chain-of-Thought = showing your work; separation = easier to improve later.

Structured Input/Output Contracts

DSPy signatures define explicit contracts between stages:

```python
class AnalyzeProblem(dspy.Signature):
    """Analyze a problem to identify its core essence."""

    problem_description = dspy.InputField()
    core_challenge = dspy.OutputField()
    key_characteristics = dspy.OutputField()
    desired_outcome = dspy.OutputField()
```

Listing 6: Example DSPy signature defining stage interface

These contracts serve multiple purposes:

1. **Type Safety:** Clear expectations for each stage

2. **Documentation:** Self-documenting interfaces
3. **Optimization:** DSPy can optimize prompts while respecting contracts
4. **Composability:** Modules can be recombined if contracts match

High Token Budget

CDSM uses a 45,000 token context window. This generous allocation reflects a fundamental trade-off: we prioritize solution quality over API cost. Each stage needs room for:

- Detailed reasoning chains
- Rich domain knowledge
- Multiple solution iterations
- Comprehensive synthesis

For production systems with cost constraints, token usage could be optimized through:

- Retrieval-augmented generation (fetch domain knowledge on-demand)
- Prompt compression techniques
- Selective Chain-of-Thought (only for difficult cases)

Example: Customer Churn Reduction

Let's trace a complete execution to illustrate the system's behavior.

User Problem

"How do I reduce customer churn in my SaaS product?"

Stage 1: Problem Analysis

Core Challenge: Maintaining sustained engagement with voluntary participants over time.

Key Characteristics:
- Voluntary participation (customers can leave anytime)
- Early-stage vulnerability (highest churn in first 90 days)
- Competing alternatives (numerous substitute products)
- Value perception gap (understanding vs. experiencing value)

Desired Outcome: Customers remain engaged long enough to reach "aha moments" and build habits around the product.

Stage 2: Domain Mapping

Identified Domains:

Domain	Relevance
Biological Immune Systems	Maintaining defense against constant threats
Ecosystem Resilience	Systems that persist despite perturbations
Social Network Dynamics	Why people stay in communities
Addiction Psychology	Mechanisms that create compulsive behavior
Jazz Improvisation	Keeping audience engaged through unpredictability

Table 2: Analogous domains identified for customer churn problem

Stage 3: Solution Extraction

From Immune Systems:
- **Domain Solution:** Immune memory and rapid response to threats
- **Mapping:** Treat churn signals like pathogens—detect early, respond fast
- **Implementation:** Build an "early warning system" tracking engagement drops; have automated re-engagement sequences ready

From Ecosystem Resilience:
- **Domain Solution:** Diversity and redundancy create stability
- **Mapping:** Single-feature dependence = fragility; multiple value streams = resilience
- **Implementation:** Ensure customers find value through multiple features; build redundant pathways to success

From Social Networks:

- **Domain Solution:** Strong weak ties—many shallow connections create network effects
- **Mapping:** Customer-to-customer connections reduce churn more than customer-to-company connections
- **Implementation:** Facilitate user communities; enable collaboration features; showcase user success stories

From Addiction Psychology:

- **Domain Solution:** Variable rewards on fixed schedules create habit loops
- **Mapping:** Predictable value delivery + unpredictable bonuses = engagement
- **Implementation:** Regular feature releases + surprise delighters; streak tracking; unlockable achievements

From Jazz Improvisation:

- **Domain Solution:** Balance structure with surprise; actively respond to audience energy
- **Mapping:** Rigid product experiences feel stale; adaptive experiences feel personal
- **Implementation:** Personalization engines; A/B test with each user; respond to usage patterns

Stage 4: Synthesis

Common Patterns:

1. **Proactive monitoring** — All domains emphasize early detection (immune response, ecosystem sensing)
2. **Relationship building** — Whether neural connections or social ties, relationships create stickiness
3. **Adaptive responses** — Rigid systems fail; flexible systems persist (jazz, ecosystems)

Unique Insights:

- Jazz improvisation suggests product should "listen" to user behavior and adapt in real-time
- Immune systems suggest "memory" of past churn signals improves future predictions

Prioritized Recommendations:

1. **Build an early warning system** (Immune Systems)
   - Track engagement decay patterns
   - Automate intervention triggers
   - Personalize re-engagement messaging
2. **Create variable reward structures** (Addiction Psychology)
   - Implement streak tracking
   - Add surprise bonuses
   - Use gamification judiciously
3. **Foster user communities** (Social Networks)

- Enable peer connections
- Facilitate collaboration
- Showcase user success publicly

Performance Characteristics

The system's performance can be characterized by several key metrics, as shown in Table 3.

Metric	Value
Latency	30-90 seconds per problem
API Calls	4 + N (where N = domains, typically 5-7)
Token Usage	15,000 + 5,000N tokens
Cost	$0.30 - $1.20 per problem

Table 3: Performance characteristics of CDSM pipeline

Detailed Cost Analysis.

Token usage scales linearly with the number of domains:

$$\text{tokens} = 15000 + 5000N \tag{1}$$

where N represents the number of domains analyzed (typically 5-7).

The number of API calls follows a similar pattern:

$$\text{calls} = 4 + N \tag{2}$$

Cost estimation for Claude Sonnet 4.5 (at $3 per million input tokens and $15 per million output tokens):

$$\begin{aligned} \text{cost} &= \text{input cost} + \text{output cost} \\ &\approx (15000 + 5000N) \times 3 \times 10^{-6} + \text{output tokens} \times 15 \times 10^{-6} \\ &\approx \$0.30 - \$1.20 \text{ per problem} \end{aligned} \tag{3}$$

For context, this represents the price of 1-4 cups of coffee for potentially business-transforming insights.

Extensions and Future Work

Retrieval-Augmented Generation

Current implementation relies entirely on the model's parametric knowledge. Adding retrieval would enable:

- Access to domain-specific textbooks and papers
- Up-to-date information on emerging domains
- Deeper technical details for solution extraction

Parallel Solution Extraction

Stage 3 currently processes domains sequentially. Parallel extraction would reduce latency by N times while requiring coordination logic:

```python
import asyncio

async def extract_parallel(domains):
    tasks = [
        self.solution_extractor(
            problem, challenge, domain
        )
        for domain in domains
    ]
    return await asyncio.gather(*tasks)
```

Listing 7: Parallel solution extraction implementation

This would modify Equation 2 to reflect concurrent execution, significantly reducing wall-clock time.

Optimization with DSPy Compilers

DSPy supports automatic prompt optimization through techniques like:

- **BootstrapFewShot:** Generate training examples from the model itself
- **MIPRO:** Optimize prompts using labeled data
- **Ensemble:** Combine multiple prompting strategies

For production deployment, compiling the pipeline could improve both quality and efficiency without changing the architecture described in Section 2.

Domain Specialization

Some domains (biology, physics, psychology) appear frequently in effective analogies. Training domain-specific modules or maintaining domain knowledge bases could improve solution depth.

User Feedback Loop

Currently, CDSM produces recommendations without learning from outcomes. Adding feedback mechanisms would enable:

- Rating solution quality
- Tracking implementation success
- Iterative refinement of domain selection heuristics

Related Work

Analogical Reasoning Systems: Prior work in AI analogical reasoning (Gentner's Structure Mapping Engine, Hofstadter's Copycat) focused on formal representation and mapping. CDSM adopts a more pragmatic approach, leveraging LLM world knowledge rather than building explicit domain models.

Problem Abstraction: The core challenge extraction parallels work in problem reformulation and abstract planning. By treating abstraction as a prompted behavior rather than a hardcoded algorithm, CDSM gains flexibility at the cost of formal guarantees.

Creative Ideation Tools: Systems like SCAMPER and TRIZ provide structured frameworks for creative thinking. CDSM can be viewed as automating aspects of these methodologies while adding LLM-powered domain knowledge.

Conclusion

The Cross-Domain Solution Matcher demonstrates how large language models can be orchestrated to perform sophisticated cognitive tasks through careful architectural design. By decomposing analogical reasoning into explicit stages (as detailed in Table 1), maintaining clear interfaces between components (see Listing 6), and forcing step-by-step reasoning, CDSM achieves both quality and interpretability.

The system's true value lies not in replacing human creativity, but in **augmenting** it—surfacing unexpected connections, providing domain expertise, and generating actionable starting points. A designer using CDSM gains a thinking partner that can draw on diverse knowledge to illuminate new solution paths.

From first principles, analogical reasoning succeeds when:
1. Problems are understood structurally rather than superficially
2. Domain knowledge is vast and varied
3. Mappings preserve relational structure
4. Synthesis identifies meta-patterns

CDSM operationalizes each principle through modular, optimizable components. As LLMs continue to improve and DSPy's optimization capabilities mature, systems like CDSM represent a promising path toward AI-augmented problem solving.

Final TL;DR: CDSM breaks down "thinking across domains" into 4 clear steps, uses Chain-of-Thought to show its work, and costs $1 to potentially unlock million-dollar insights. The future of creative AI isn't about replacing humans—it's about giving them better thinking tools.

—