KURSE: A Neuro-Symbolic Inference Engine

Technical Report on Knowledge-Unified Reasoning and Semantic Engine
with Application to BeeKurse E-Commerce Platform

Varshith Kada Vishnu Teja Himesh Anant Siripuru Abhiram
Abhinav Goyal Bhuvan Bhedika Kunjan Manoj

December 4, 2025

Abstract

KURSE (Knowledge Utilization, Retrieval and Summarization Engine) is a neuro-symbolic inference engine that combines the flexibility of neural networks with the precision of symbolic reasoning. Built on a tri-store architecture integrating SQL databases, vector databases, and knowledge graphs, KURSE enables sophisticated reasoning over unstructured data through automated knowledge extraction and ontology-driven inference. This report presents the complete technical architecture of KURSE, detailing the input processing pipeline, vector database ingestion, knowledge graph construction with N-ary relations, and ontology-based reasoning mechanisms. We demonstrate two applications: *Alibi Breaker* for constraint satisfaction problems and *Hidden Contagion* for graph traversal analysis. Additionally, we present BeeKurse, an adaptation of the KURSE framework for conversational e-commerce, featuring a novel query classification system, semantic search orchestration, and intelligent product recommendation through knowledge graph traversal. The report provides comprehensive technical details including algorithms, data structures, and implementation specifics.

Contents

1 Introduction

The proliferation of unstructured data in modern applications necessitates intelligent systems capable of not only storing and retrieving information but also reasoning over it to derive meaningful insights. Traditional database systems excel at structured queries but fall short when semantic understanding is required. Conversely, pure neural approaches, while powerful for pattern recognition, often lack the interpretability and logical consistency demanded by many applications.

KURSE addresses this challenge through a **neuro-symbolic approach**, combining:

- **Neural Components**: Large Language Models (LLMs) for natural language understanding, entity extraction, and semantic parsing
- **Symbolic Components**: Knowledge graphs with formal ontologies for structured reasoning, constraint validation, and logical inference

1.1 The Tri-Store Architecture

At the core of KURSE lies a tri-store architecture that leverages the complementary strengths of three storage paradigms:

1. **SQL Database**: Handles structured metadata, transactional data, and relational queries with ACID guarantees
2. **Vector Database (VDB)**: Enables semantic similarity search through high-dimensional embeddings, supporting fuzzy matching and approximate nearest neighbor queries
3. **Knowledge Graph (KG)**: Represents entities and their relationships in a graph structure, enabling traversal-based reasoning and ontology-driven inference

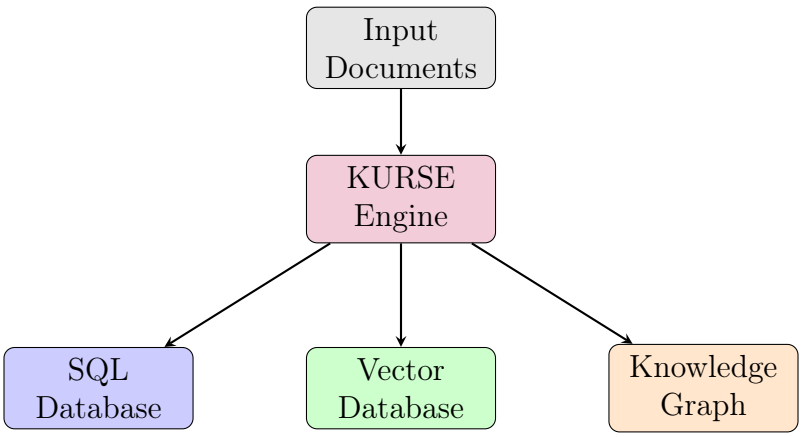

Figure 1: KURSE Tri-Store Architecture Overview

1.2 Design Philosophy

KURSE is designed as an **adaptable framework** rather than a fixed solution. The core architecture can be customized for different domains by:

- Defining domain-specific ontologies
- Configuring entity types and relation schemas
- Adjusting reasoning strategies and inference rules
- Tuning similarity thresholds and scoring parameters

This adaptability is demonstrated through two distinct applications presented in this report: *Alibi Breaker* for investigative reasoning and *BeeKurse* for e-commerce recommendations.

2 System Architecture

2.1 High-Level Overview

The KURSE system processes input documents through a multi-stage pipeline, transforming unstructured text into structured, queryable knowledge representations.

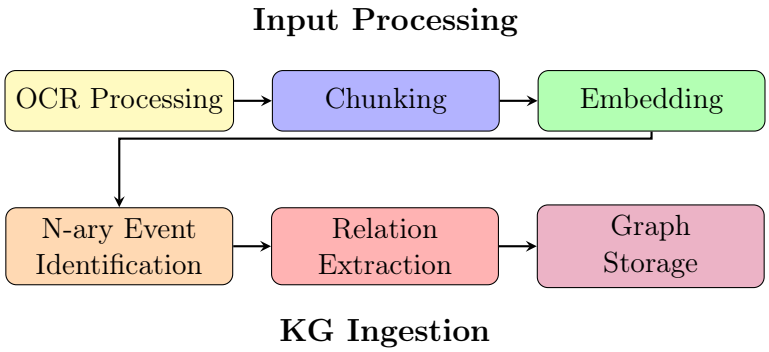

Figure 2: KURSE Processing Pipeline

2.2 Component Overview

2.2.1 OCR Module

The OCR module utilizes **olmOCR**, a vision-language model based on Qwen 7B, to extract text from images and PDF documents. Key features include:

- Multi-page PDF processing with page-level metadata tracking
- Image preprocessing for enhanced text recognition
- Structured output preserving document layout information

2.2.2 Vector Database (Qdrant)

KURSE employs Qdrant as its vector database, storing:

- Document chunk embeddings for semantic search
- Entity type embeddings for standardization
- Relation type embeddings for schema matching

2.2.3 Knowledge Graph (Memgraph)

Memgraph serves as the graph database, supporting:

- Cypher query language for graph operations
- MERGE operations for idempotent entity creation
- Property graphs with rich metadata on nodes and edges

2.2.4 SQL Database

The SQL layer manages:

- Document metadata and processing status
- User information and session data
- Transaction logs and audit trails

3 Input Processing

3.1 OCR with olmOCR

The input processing pipeline begins with optical character recognition using olmOCR, a specialized model for document understanding.

3.1.1 Model Architecture

olmOCR is built on the Qwen 7B vision-language model, fine-tuned for:

- Dense text recognition in documents
- Table structure understanding
- Handwritten text interpretation
- Multi-language support

3.1.2 Processing Flow

3.2 Document ID Generation

Each document is assigned a unique identifier using a combination of content hashing and temporal information:

Listing 1: Document ID Generation

```python
import hashlib
from datetime import datetime

def generate_document_id(content: bytes) -> str:
    content_hash = hashlib.md5(content).hexdigest()[:8]
    timestamp = datetime.now().strftime("%Y%m%d%H%M%S")
    return f"doc_{content_hash}_{timestamp}"
```

Algorithm 1 OCR Processing Pipeline

Require: Input document D (PDF or image)
Ensure: Extracted text with metadata T
if D is PDF **then**
  $pages \leftarrow \text{ExtractPages}(D)$
  for each $page$ in $pages$ **do**
    $text_i \leftarrow \text{olmOCR}(page)$
    $T \leftarrow T \cup \{(text_i, page\_no)\}$
  end for
else
  $T \leftarrow \text{olmOCR}(D)$
end if
return T

This approach ensures:

- **Uniqueness**: Timestamp prevents collisions for identical content
- **Traceability**: Hash prefix enables content verification
- **Sortability**: Timestamp component allows chronological ordering

3.3 Chunking Algorithm

Raw text is segmented into manageable chunks for processing and storage. The chunking algorithm balances:

- **Context preservation**: Maintaining semantic coherence within chunks
- **Size optimization**: Fitting within LLM context windows
- **Overlap strategy**: Ensuring boundary information is not lost

3.3.1 Algorithm Parameters

Parameter	Value	Purpose
`CHUNK_SIZE`	1000 chars	Maximum chunk length
`CHUNK_OVERLAP`	200 chars	Overlap between consecutive chunks
`SENTENCE_BOUNDARY`	True	Respect sentence boundaries

Table 1: Chunking Configuration Parameters

3.3.2 Sentence Boundary Detection

The chunking algorithm attempts to break at sentence boundaries when possible:

The sentence boundary detection uses punctuation markers (., !, ?) followed by whitespace to identify natural break points.

Algorithm 2 Semantic Chunking Algorithm

Require: Text T, chunk_size S, overlap O
Ensure: List of chunks C

$C \leftarrow []$
$start \leftarrow 0$
while $start < len(T)$ **do**
  $end \leftarrow \min(start + S, len(T))$
  if $end < len(T)$ **then**
    $boundary \leftarrow \text{FindSentenceBoundary}(T[start : end])$
    if $boundary > S \times 0.7$ **then** ▷ Avoid tiny chunks
      $end \leftarrow start + boundary$
    end if
  end if
  C.append$(T[start : end])$
  $start \leftarrow end - O$
end while
return C

4 Vector Database Ingestion

4.1 Embedding Generation

KURSE utilizes NVIDIA's **NVCLIP** model for generating document embeddings, producing 1024-dimensional vectors optimized for semantic similarity.

4.1.1 NVCLIP Architecture

- **Model**: NVIDIA CLIP variant trained on diverse document corpora
- **Dimension**: 1024-dimensional dense vectors
- **Normalization**: L2-normalized for cosine similarity computation

4.1.2 Embedding Pipeline

Listing 2: Embedding Generation

```python
from nvclip import NVCLIPEncoder

encoder = NVCLIPEncoder(model="nvclip-1024")

def embed_chunk(chunk: str) -> List[float]:
    """Generate 1024-dim embedding for text chunk."""
    embedding = encoder.encode(chunk)
    return embedding.tolist()  # 1024-dimensional vector
```

4.2 Qdrant Storage Schema

Embeddings are stored in Qdrant with rich metadata payloads:

Listing 3: Qdrant Point Structure

```
{
  "id": "chunk_abc123_001",
  "vector": [0.123, -0.456, ...],  // 1024 dimensions
  "payload": {
    "document_id": "doc_a1b2c3d4_20240101120000",
    "chunk_id": "chunk_001",
    "chunk_index": 0,
    "page_no": 1,
    "text": "Original chunk text...",
    "created_at": "2024-01-01T12:00:00Z"
  }
}
```

4.3 Collection Configuration

Listing 4: Qdrant Collection Setup

```
from qdrant_client import QdrantClient
from qdrant_client.models import VectorParams, Distance

client = QdrantClient(host="localhost", port=6333)

client.create_collection(
    collection_name="document_chunks",
    vectors_config=VectorParams(
        size=1024,
        distance=Distance.COSINE
    )
)
```

5 Knowledge Graph Fundamentals

5.1 Graph Data Model

The KURSE knowledge graph employs a property graph model with three primary node categories:

5.1.1 Node Types

1. **Entities**: Real-world objects, people, places, or concepts
   - Examples: Person, Organization, Location, Event
   - Properties: name, type, aliases, metadata
2. **Properties**: Attributes that describe entities
   - Examples: color, size, material, date
   - Can be categorical or continuous

3. **Literals**: Concrete values
    - Examples: “red”, 42, “2024-01-01”
    - Typed values with units where applicable

5.1.2 Edge Types

Relationships connect nodes with semantic meaning:

- **Object Properties**: Entity-to-Entity relationships (e.g., WORKS_FOR, LOCATED_IN)
- **Data Properties**: Entity-to-Literal relationships (e.g., HAS_AGE, HAS_NAME)

5.2 N-ary Relations

Traditional binary relations (subject-predicate-object triplets) are insufficient for representing complex real-world scenarios. KURSE implements **N-ary relations** to capture multi-participant events.

5.2.1 The Problem with Binary Relations

Consider the statement: “John sold a car to Mary for $5000 on January 1st.”

Binary representation would require:

- (John, SOLD, Car)
- (Mary, BOUGHT, Car)
- (Car, HAS_PRICE, $5000)
- (Transaction, HAS_DATE, January 1st)

This loses the connection between all elements of the single transaction.

5.2.2 N-ary Solution

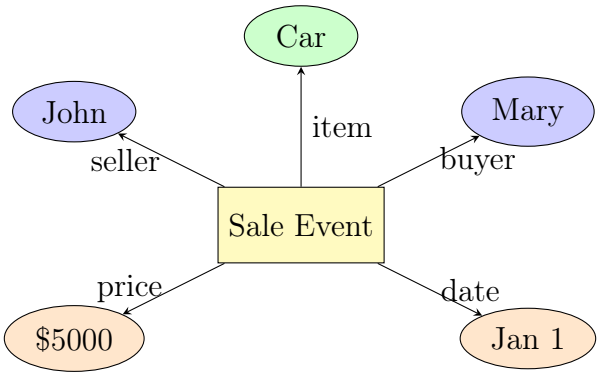

Figure 3: N-ary Relation Example: Sale Event with Multiple Participants

The N-ary approach:

1. Creates an **Event Node** representing the composite action
2. Connects all participants through **role-labeled edges**
3. Preserves the semantic unity of the event
4. Enables complex queries across all event participants

5.3 Graph Structure in Memgraph

Listing 5: Cypher Schema for N-ary Events

```
// Create Event Node
CREATE (e:Event {
    id: "event_001",
    type: "Sale",
    source_chunk: "chunk_abc123_001",
    confidence: 0.95
})

// Connect Participants
MATCH (e:Event {id: "event_001"})
MATCH (seller:Person {name: "John"})
MATCH (buyer:Person {name: "Mary"})
MATCH (item:Product {name: "Car"})
CREATE (e)-[:HAS_SELLER]->(seller)
CREATE (e)-[:HAS_BUYER]->(buyer)
CREATE (e)-[:HAS_ITEM]->(item)
CREATE (e)-[:HAS_PRICE]->(:Literal {value: 5000, unit: "USD"})
```

6 Ontology and Reasoning

This section presents the theoretical foundations of ontology-based reasoning as employed in KURSE.

6.1 What is an Ontology?

An **ontology** is a formal, explicit specification of a shared conceptualization. In the context of KURSE:

- **Formal**: Machine-readable with precise semantics
- **Explicit**: Clearly defined concepts and constraints
- **Shared**: Common understanding across system components
- **Conceptualization**: Abstract model of the domain

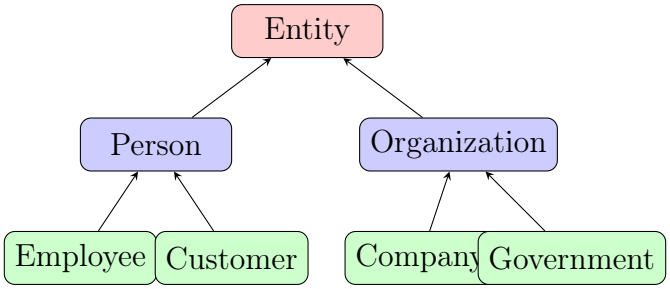

Figure 4: Class Hierarchy Example

6.2 Class Hierarchies (subClassOf)

Ontologies define hierarchical relationships between concepts using the `subClassOf` relation:

Inheritance Properties:

- Subclasses inherit all properties of parent classes
- Instances of subclasses are also instances of all ancestor classes
- Enables polymorphic querying across class hierarchies

6.3 Ontology Axioms

KURSE supports several types of axioms for reasoning:

6.3.1 Transitivity

A relation R is **transitive** if:

$$\forall x, y, z : R(x, y) \land R(y, z) \Rightarrow R(x, z)$$

Example: The `LOCATED_IN` relation is transitive.

- If "Building A is located in City X" and "City X is located in Country Y"
- Then "Building A is located in Country Y" can be inferred

Listing 6: Transitive Query in Cypher

```
// Find all locations containing entity (transitive closure)
MATCH (e:Entity {name: "Building␣A"})
      -[:LOCATED_IN*1..]->(loc:Location)
RETURN loc.name
```

6.3.2 Symmetry

A relation R is **symmetric** if:

$$\forall x, y : R(x, y) \Rightarrow R(y, x)$$

Example: The `SIBLING_OF` relation is symmetric.

- If "Alice is sibling of Bob"
- Then "Bob is sibling of Alice" is automatically true

6.3.3 Equivalence

Two classes A and B are **equivalent** if:

$$A \equiv B \Leftrightarrow (A \sqsubseteq B) \wedge (B \sqsubseteq A)$$

This allows treating instances of one class as instances of the other.

6.4 First-Order Logic Mappings

Ontological constraints map to first-order logic (FOL) for formal reasoning:

Ontology Construct	FOL Representation
`subClassOf(A, B)`	$\forall x : A(x) \Rightarrow B(x)$
`domain(R, C)`	$\forall x, y : R(x, y) \Rightarrow C(x)$
`range(R, C)`	$\forall x, y : R(x, y) \Rightarrow C(y)$
`transitive(R)`	$\forall x, y, z : R(x, y) \wedge R(y, z) \Rightarrow R(x, z)$
`symmetric(R)`	$\forall x, y : R(x, y) \Rightarrow R(y, x)$

Table 2: Ontology to First-Order Logic Mappings

6.5 Domain and Range Constraints

Domain and range constraints restrict the valid types for relation participants:

- **Domain**: Specifies the valid type for the subject of a relation
- **Range**: Specifies the valid type for the object of a relation

Example: For relation `WORKS_FOR`:

- Domain: `Person` (only persons can work)
- Range: `Organization` (can only work for organizations)

Listing 7: Domain/Range Validation

```
def validate_relation(subject, predicate, object, ontology):
    """Validate relation against domain/range constraints."""
    domain = ontology.get_domain(predicate)
    range_ = ontology.get_range(predicate)

    if not is_instance_of(subject, domain):
        raise ValidationError(
            f"Subject {subject} not in domain {domain}"
        )
    if not is_instance_of(object, range_):
        raise ValidationError(
            f"Object {object} not in range {range_}"
        )
    return True
```

6.6 Inference Strategies

KURSE employs two inference strategies:

6.6.1 Forward Chaining

1. Start with known facts
2. Apply inference rules to derive new facts
3. Continue until no new facts can be derived
4. **Use case**: Precomputing all derivable knowledge at ingestion time

6.6.2 Backward Chaining

1. Start with a query (goal)
2. Work backward to find supporting facts
3. Apply rules in reverse to verify the goal
4. **Use case**: Query-time inference for specific questions

7 Knowledge Graph Ingestion Pipeline

The KG ingestion pipeline is the core of KURSE's knowledge extraction capabilities. It operates in two phases with a streaming bridge architecture.

7.1 Phase 1: N-ary Event Identification

The first phase identifies complex events that involve multiple participants and require N-ary representation.

7.1.1 Event Criteria

An N-ary event is created when a chunk contains:

1. **Three or more participants**: Multiple entities involved in a single action
2. **Critical metadata**: Temporal information, locations, or quantitative data essential to understanding the event
3. **Semantic unity**: The participants are connected by a single coherent action or state

7.1.2 LLM-Based Event Detection

Phase 1 System Prompt (Simplified)

You are an expert at identifying complex events in text. Analyze the given text chunk and identify any events that involve 3 or more participants or have critical metadata that links multiple entities.
For each event, provide:

- event_type: The category of event (Transaction, Meeting, Communication, etc.)
- participants: List of entities involved with their roles
- metadata: Critical information (dates, amounts, locations)
- confidence: Your confidence in this identification (0-1)

Only identify events where representing as binary relations would lose important connections.

7.1.3 Event Registry

Identified events are stored in an event registry with VDB embeddings:

Listing 8: Event Data Structure

```python
@dataclass
class NaryEvent:
    event_id: str
    event_type: str
    participants: List[Participant]
    metadata: Dict[str, Any]
    source_chunk_id: str
    source_document_id: str
    confidence: float
    embedding: List[float]  # 1024-dim

@dataclass
class Participant:
    entity_name: str
    entity_type: str
    role: str  # seller, buyer, witness, etc.
```

7.1.4 Manifest System

The manifest system tracks processing state and provides context for Phase 2:

Listing 9: Manifest Structure

```
{
  "document_id": "doc_abc123",
  "chunk_id": "chunk_001",
  "page_no": 1,
  "events_identified": [
    {
```

```
      "event_id": "evt_001",
      "event_type": "Transaction",
      "participant_count": 4
    }
  ],
  "processing_status": "phase1_complete",
  "timestamp": "2024-01-01T12:00:00Z"
}
```

7.2 Phase 2: Relation Extraction

The second phase extracts individual relations (triplets) from the text, including both simple binary relations and connections to N-ary events.

7.2.1 Triplet Extraction with LLM

Algorithm 3 Relation Extraction Algorithm

Require: Chunk C, Manifest M
Ensure: List of standardized triplets T

$raw\_triplets \leftarrow \text{LLMExtract}(C, M)$
for each *triplet* in $raw\_triplets$ **do**
  $verified \leftarrow \text{VerifyAgainstSource}(triplet, C)$
  if $verified$ **then**
    $std\_triplet \leftarrow \text{StandardizeToOntology}(triplet)$
    T.append($std\_triplet$)
  end if
end for
return T

7.2.2 RawTriplet Data Structure

Listing 10: RawTriplet Definition

```python
from pydantic import BaseModel
from typing import Optional

class RawTriplet(BaseModel):
    subject: str            # Entity name
    subject_type: str       # Raw type from LLM
    predicate: str          # Relation name
    object: str             # Entity or literal
    object_type: str        # Raw type from LLM
    confidence: float       # Extraction confidence
    source_span: Optional[str]  # Text span supporting triplet
```

7.2.3 Verification Against Source Text

Each extracted triplet is verified against the source text to reduce hallucinations:

Listing 11: Triplet Verification

```
def verify_triplet(triplet: RawTriplet, source_text: str) -> bool:
    """Verify triplet is supported by source text."""
    verification_prompt = f"""
    Source text: {source_text}

    Claimed fact: {triplet.subject} {triplet.predicate} {triplet.
    object}

    Is this fact directly supported by the source text?
    Answer only YES or NO with brief justification.
    """

    response = llm.generate(verification_prompt)
    return response.startswith("YES")
```

7.2.4 Standardization to Ontology

Raw entity types and relations are standardized using VDB similarity matching:

Listing 12: Type Standardization

```
TYPE_THRESHOLD = 0.25       # Minimum similarity for type match
RELATION_THRESHOLD = 0.20   # Minimum similarity for relation match

def standardize_type(raw_type: str, types_vdb: QdrantClient) ->
    str:
    """Match raw type to ontology types using VDB."""
    embedding = encoder.encode(raw_type)

    results = types_vdb.search(
        collection_name="entity_types",
        query_vector=embedding,
        limit=1
    )

    if results[0].score >= TYPE_THRESHOLD:
        return results[0].payload["type_name"]
    else:
        return "Unknown"  # Flag for manual review

def standardize_relation(raw_rel: str, relations_vdb: QdrantClient
    ) -> str:
    """Match raw relation to ontology relations using VDB."""
    embedding = encoder.encode(raw_rel)

    results = relations_vdb.search(
        collection_name="relation_types",
```

```
        query_vector=embedding,
        limit=1
    )

    if results[0].score >= RELATION_THRESHOLD:
        return results[0].payload["relation_name"]
    else:
        return raw_rel  # Keep raw if no match
```

7.2.5 Threshold Selection Rationale

- **TYPE_THRESHOLD = 0.25**: Lower threshold allows capturing entities even with varied naming. Types are more critical to get right, so unmatched types are flagged.
- **RELATION_THRESHOLD = 0.20**: Even lower threshold for relations since natural language expresses relations in many ways. Novel relations are preserved.

7.3 Streaming Bridge Architecture

The bridge connects Phase 1 and Phase 2 through an asynchronous queue system:

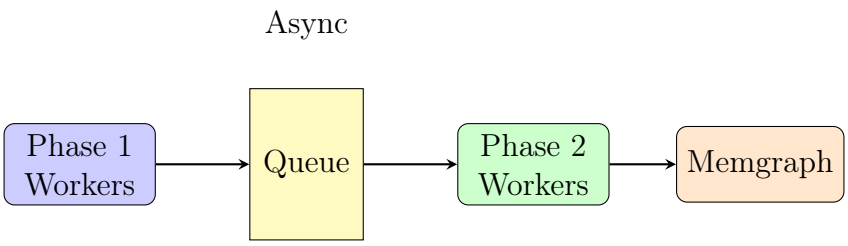

Figure 5: Streaming Bridge Architecture

Listing 13: Bridge Implementation

```
import asyncio
from asyncio import Queue

class StreamingBridge:
    def __init__(self, num_workers: int = 4):
        self.queue = Queue()
        self.num_workers = num_workers

    async def phase1_producer(self, chunks: List[str]):
        """Process chunks and push to queue."""
        for chunk in chunks:
            events = await identify_nary_events(chunk)
            manifest = create_manifest(chunk, events)
            await self.queue.put((chunk, manifest))

        # Signal completion
        for _ in range(self.num_workers):
```

```
            await self.queue.put(None)

    async def phase2_worker(self, worker_id: int):
        """Consume from queue and process."""
        while True:
            item = await self.queue.get()
            if item is None:
                break

            chunk, manifest = item
            triplets = await extract_relations(chunk, manifest)
            await store_to_memgraph(triplets)

    async def run(self, chunks: List[str]):
        """Execute pipeline."""
        producer = asyncio.create_task(
            self.phase1_producer(chunks)
        )
        workers = [
            asyncio.create_task(self.phase2_worker(i))
            for i in range(self.num_workers)
        ]

        await producer
        await asyncio.gather(*workers)
```

7.4 Memgraph Storage

7.4.1 Cypher MERGE Operations

Entities and relations are stored using MERGE operations for idempotency:

Listing 14: Entity and Relation Storage

```
// Store Entity (idempotent)
MERGE (e:Entity {name: $name})
ON CREATE SET
    e.type = $type,
    e.created_at = timestamp(),
    e.source_documents = [$document_id]
ON MATCH SET
    e.source_documents = e.source_documents + $document_id

// Store Relation
MATCH (s:Entity {name: $subject})
MATCH (o:Entity {name: $object})
MERGE (s)-[r:$predicate]->(o)
ON CREATE SET
    r.source_chunk = $chunk_id,
    r.confidence = $confidence,
    r.created_at = timestamp()
```

7.4.2 Metadata Tracking

Each node and edge tracks provenance information:

Listing 15: Node Metadata

```
{
  "name": "John Smith",
  "type": "Person",
  "source_documents": ["doc_001", "doc_002"],
  "source_chunks": ["chunk_001", "chunk_015"],
  "page_numbers": [1, 5],
  "first_seen": "2024-01-01T12:00:00Z",
  "last_updated": "2024-01-15T08:30:00Z",
  "mention_count": 15
}
```

7.4.3 Conflict Resolution

When duplicate or conflicting information is encountered:

1. **Entity Merging**: Entities with same name are merged; properties are unioned
2. **Relation Confidence**: Multiple extractions increase confidence score
3. **Source Tracking**: All source documents are preserved for traceability
4. **Version History**: Timestamps track when information was added/modified

8 Verification and Validation

8.1 Chunk Verification

Before triplets are stored, they undergo LLM-based fact-checking:

Listing 16: Chunk Verification Pipeline

```
async def verify_chunk_extractions(
    chunk: str,
    triplets: List[Triplet]
) -> List[Triplet]:
    """Verify extracted triplets against source chunk."""

    verified = []
    for triplet in triplets:
        prompt = f"""
        Source text: {chunk}

        Verify this extracted fact:
        Subject: {triplet.subject} (type: {triplet.subject_type})
        Relation: {triplet.predicate}
        Object: {triplet.object} (type: {triplet.object_type})

```

```
        Is this fact:
        1. Explicitly stated in the text?
        2. Reasonably inferable from the text?
        3. Not supported by the text?

        Answer with number and brief explanation.
        """

            response = await llm.generate(prompt)
            if response.startswith("1") or response.startswith("2"):
                verified.append(triplet)

    return verified
```

8.2 Ontology Feasibility Checking

Extracted relations are validated against ontology constraints:

Listing 17: Ontology Validation

```
def check_ontology_feasibility(
    triplet: Triplet,
    ontology: Ontology
) -> Tuple[bool, str]:
    """Check if triplet is feasible under ontology."""

    # Check domain constraint
    if triplet.predicate in ontology.relations:
        expected_domain = ontology.get_domain(triplet.predicate)
        if not ontology.is_subclass(triplet.subject_type,
            expected_domain):
            return False, f"Domain violation: {triplet.
                subject_type} not in {expected_domain}"

    # Check range constraint
        expected_range = ontology.get_range(triplet.predicate)
        if not ontology.is_subclass(triplet.object_type,
            expected_range):
            return False, f"Range violation: {triplet.object_type}
                not in {expected_range}"

    return True, "Valid"
```

9 Applications

KURSE's adaptable architecture enables diverse applications. We present two case studies demonstrating the system's capabilities.

9.1 Alibi Breaker: Constraint Satisfaction

Alibi Breaker is an investigative reasoning application that identifies contradictions in statements and alibis.

9.1.1 Problem Definition

Given a set of statements from multiple sources, identify:

- Temporal inconsistencies (impossible timelines)
- Spatial contradictions (impossible locations)
- Logical impossibilities (mutually exclusive claims)

9.1.2 Approach

1. Extract temporal and spatial entities from statements
2. Build constraint graph with must-hold relationships
3. Apply constraint propagation to detect violations
4. Present contradictions with supporting evidence

9.1.3 Example Query

Listing 18: Alibi Contradiction Query

```
// Find persons with contradictory locations at same time
MATCH (p:Person)-[:WAS_AT]->(l1:Location)
      -[:AT_TIME]->(t:Time)
MATCH (p)-[:WAS_AT]->(l2:Location)
      -[:AT_TIME]->(t)
WHERE l1 <> l2
RETURN p.name, l1.name, l2.name, t.value
```

9.2 Hidden Contagion: Graph Traversal

Hidden Contagion traces the spread of information, influence, or contact through networks.

9.2.1 Problem Definition

Given a starting entity and a time window:

- Identify all entities that came into contact (directly or transitively)
- Compute the propagation path and timeline
- Assess exposure levels based on contact duration/frequency

9.2.2 Approach

1. Start from patient zero / source entity
2. Traverse contact relationships with temporal constraints
3. Apply transitive closure within time bounds
4. Rank contacts by exposure metrics

9.2.3 Example Query

Listing 19: Contagion Trace Query

```
// Trace all contacts within time window
MATCH path = (source:Person {name: $patient_zero})
             -[:CONTACTED*1..5]-(contact:Person)
WHERE all(r IN relationships(path)
          WHERE r.timestamp >= $start_time
          AND r.timestamp <= $end_time)
RETURN contact.name, length(path) as degrees,
       [r IN relationships(path) | r.timestamp] as timeline
ORDER BY degrees
```

10 BeeKurse: E-Commerce Application

BeeKurse adapts the KURSE framework for conversational e-commerce, enabling natural language product search and intelligent recommendations.

10.1 Platform Overview

BeeKurse is a **conversational e-commerce platform** designed to:

- Enable natural language product queries
- Support small vendors through OCR-based product ingestion
- Provide intelligent recommendations via knowledge graph traversal
- Maintain user context across conversation turns

10.2 Adapting KURSE for Commerce

BeeKurse makes strategic modifications to the KURSE architecture:

10.2.1 Dropped Strict Ontology

Unlike investigative applications requiring logical rigor, e-commerce benefits from:

- **Flexible relations**: "goes well with" rather than formal predicates
- **Creative associations**: Trend-based and style-based connections
- **Fuzzy matching**: Approximate product similarity over exact types

10.2.2 VDB for Relations

Relations are stored in the vector database rather than a formal ontology:

Listing 20: Relation Storage in VDB

```
# Instead of ontology-defined relations
relations_collection = {
    "ALSO_BOUGHT": [],       # Co-purchase patterns
    "SIMILAR_TO": [],        # Style/feature similarity
    "COMPLEMENTS": [],       # Complementary products
    "TRENDING_WITH": []      # Trend associations
}
```

11 Query Classification System

BeeKurse employs a sophisticated query classification system to route user requests to appropriate handlers.

11.1 Query Types

The system recognizes five query types:

Type	Purpose	Example
SEARCH	Product search	"Show me red dresses under $50"
DETAIL	Product info	"Tell me more about product #123"
CHAT	General chat	"What's trending this season?"
CART_ACTION	Cart modify	"Add this to my cart"
CART_VIEW	Cart view	"What's in my cart?"

Table 3: BeeKurse Query Types

11.2 LLM Parser Architecture: Strontium

The query parser, codenamed **Strontium**, uses a 269-line system prompt to achieve precise query understanding.

11.2.1 System Prompt Structure

Strontium System Prompt Overview

1. **Role Definition**: Expert e-commerce query parser
2. **Output Format**: Strict JSON schema
3. **Query Type Rules**: Decision tree for classification
4. **Property Extraction**: Weight assignment guidelines
5. **Literal Handling**: Buffer and range rules
6. **User Context**: Preference integration rules
7. **Examples**: 20+ annotated examples per query type

11.2.2 Parser Output Format

Listing 21: Parsed Query Structure

```
{
  "query_type": "SEARCH",
  "product_request": {
    "product_type": "dress",
    "properties": [
      {"name": "color", "value": "red", "weight": 1.5}
    ],
    "literals": [
      {"name": "price", "value": 50, "operator": "<=", "buffer": 10}
    ],
    "is_hq": false,
    "sort_literal": null
  },
  "user_context_applied": true
}
```

12 SEARCH Query Processing

SEARCH queries are the most complex, involving multiple retrieval strategies and scoring mechanisms.

12.1 ProductRequest Model

Listing 22: ProductRequest Pydantic Model

```
from pydantic import BaseModel
from typing import List, Optional
from enum import Enum

```

```python
class Property(BaseModel):
    name: str
    value: str
    weight: float = 1.0  # 0.5 to 2.0 scale

class Operator(str, Enum):
    EQ = "="
    LT = "<"
    LE = "<="
    GT = ">"
    GE = ">="
    BETWEEN = "between"

class Literal(BaseModel):
    name: str
    value: float
    operator: Operator
    buffer: float = 0.0  # Tolerance range

class ProductRequest(BaseModel):
    product_type: str
    properties: List[Property] = []
    literals: List[Literal] = []
    is_hq: bool = False
    sort_literal: Optional[str] = None
    subcategory: Optional[str] = None
```

12.2 Property Weights

Properties are assigned importance weights on a 0.5-2.0 scale:

Weight	Level	Indicators
2.0	Critical	Explicit emphasis ("must be", "definitely")
1.5	Important	Direct mention with context
1.0	Standard	Simple mention
0.5	Nice-to-have	Soft preferences ("preferably", "if possible")

Table 4: Property Weight Scale

12.2.1 Weight Assignment Examples

Listing 23: Weight Assignment Examples

```
// "I need a red dress" -> Standard mention
{"name": "color", "value": "red", "weight": 1.0}

```

```
// "Must be cotton, nothing synthetic" -> Critical
{"name": "material", "value": "cotton", "weight": 2.0}

// "Preferably with pockets" -> Nice-to-have
{"name": "feature", "value": "pockets", "weight": 0.5}

// "Looking for something floral, really want that pattern"
{"name": "pattern", "value": "floral", "weight": 1.5}
```

12.3 Literals with Buffers

Numeric constraints include buffers for flexibility:

Listing 24: Literal Buffer Application

```
def apply_literal_filter(products, literal: Literal):
    """Filter products with buffer tolerance."""
    effective_value = literal.value
    buffer = literal.buffer

    if literal.operator == "<=":
        max_val = effective_value + buffer
        return [p for p in products if p[literal.name] <= max_val]

    elif literal.operator == ">=":
        min_val = effective_value - buffer
        return [p for p in products if p[literal.name] >= min_val]

    elif literal.operator == "between":
        # value is tuple (min, max)
        return [p for p in products
                if literal.value[0] - buffer <= p[literal.name]
                   <= literal.value[1] + buffer]
```

12.3.1 Buffer Calculation

Listing 25: Dynamic Buffer Calculation

```
def calculate_buffer(literal_name: str, value: float) -> float:
    """Calculate appropriate buffer based on literal type."""
    buffer_rules = {
        "price": lambda v: v * 0.15,       # 15% of price
        "rating": lambda v: 0.3,            # Fixed 0.3 stars
        "size": lambda v: 1,                # +/- 1 size
        "quantity": lambda v: 0,            # Exact match
    }

    rule = buffer_rules.get(literal_name, lambda v: v * 0.1)
    return rule(value)
```

12.4 HQ Detection (Hurry Query)

HQ (Hurry Query) indicates the user wants quick results, skipping semantic search:

12.4.1 Explicit HQ Indicators

- "Show me any..."
- "Just find..."
- "Quick, I need..."
- "Whatever is available..."

12.4.2 Implicit HQ Indicators

- Very generic queries with no properties
- Queries mentioning specific product IDs
- Re-queries after failed detailed searches

Listing 26: HQ Detection Logic

```python
def detect_hq(query: str, parsed: ProductRequest) -> bool:
    """Detect if query should use fast path."""

    # Explicit indicators
    hq_phrases = ["show me any", "just find", "quick", "whatever"]
    if any(phrase in query.lower() for phrase in hq_phrases):
        return True

    # Implicit: No properties specified
    if len(parsed.properties) == 0 and len(parsed.literals) <= 1:
        return True

    # Implicit: Product ID mentioned
    if re.search(r'#\d+|product\s+\d+', query.lower()):
        return True

    return False
```

12.5 Sort Literal for Superlatives

Superlative queries trigger special sorting:

13 DETAIL Query Processing

DETAIL queries request information about specific products.

Superlative	Sort Field	Direction
“cheapest”	price	ASC
“most expensive”	price	DESC
“highest rated”	rating	DESC
“most popular”	sales_count	DESC
“newest”	created_at	DESC

Table 5: Superlative to Sort Mapping

13.1 Product ID Resolution

Listing 27: Product ID Resolution

```python
def resolve_product_id(query: str, context: ConversationContext)
    -> str:
    """Resolve product reference to actual ID."""

    # Direct ID mention: "product #123"
    match = re.search(r'#(\d+)|product\s+(\d+)', query.lower())
    if match:
        return match.group(1) or match.group(2)

    # Contextual reference: "this one", "the first one"
    if "this" in query.lower() or "first" in query.lower():
        return context.last_shown_products[0]

    if "second" in query.lower():
        return context.last_shown_products[1]

    # Descriptive reference: "the red one"
    return resolve_by_description(query, context.
        last_shown_products)
```

13.2 Detail Query Flow

1. Resolve product ID from query
2. Fetch product from SQL database
3. Build LLM context with product details
4. Generate natural language answer

Listing 28: Detail Query Handler

```python
async def handle_detail_query(
    product_id: str,
    question: str
) -> str:
    """Handle detail query about specific product."""
```

```
    # Fetch from SQL
    product = await sql_client.fetch_product(product_id)

    # Build context
    context = f"""
    Product: {product.name}
    Price: ${product.price}
    Description: {product.description}
    Specs: {json.dumps(product.specifications)}
    Reviews Summary: {product.reviews_summary}
    """

    # Generate answer
    prompt = f"""
    Product Information:
    {context}

    User Question: {question}

    Provide a helpful, accurate answer based on the product
       information.
    """

    return await llm.generate(prompt)
```

14 CHAT and CART Queries

14.1 CHAT Query Handling

CHAT queries are general conversation not directly tied to product search:

Listing 29: Chat Query Handler

```
async def handle_chat_query(
    query: str,
    user_context: UserContext
) -> str:
    """Handle general chat queries."""

    # Build contextual prompt
    prompt = f"""
    You are a helpful e-commerce assistant for BeeKurse.

    User preferences: {user_context.preferences}
    Previous interactions: {user_context.recent_queries[-5:]}

    User message: {query}

    Provide helpful, friendly response. If the query could
    be better served by a product search, suggest that.
```

```
    """

    return await llm.generate(prompt)
```

14.2 CART Action Handling

Cart actions modify the user's shopping cart:

Listing 30: Cart Action Handler

```
class CartAction(str, Enum):
    ADD = "add"
    REMOVE = "remove"
    UPDATE_QUANTITY = "update"
    CLEAR = "clear"

async def handle_cart_action(
    action: CartAction,
    product_id: Optional[str],
    quantity: int = 1,
    user_id: str
) -> CartResponse:
    """Execute cart modification."""

    cart = await get_user_cart(user_id)

    if action == CartAction.ADD:
        cart.add_item(product_id, quantity)
    elif action == CartAction.REMOVE:
        cart.remove_item(product_id)
    elif action == CartAction.UPDATE_QUANTITY:
        cart.update_quantity(product_id, quantity)
    elif action == CartAction.CLEAR:
        cart.clear()

    await save_cart(cart)
    return CartResponse(success=True, cart=cart)
```

14.3 CART View Handling

Listing 31: Cart View Handler

```
async def handle_cart_view(user_id: str) -> str:
    """Generate cart summary for user."""

    cart = await get_user_cart(user_id)

    if cart.is_empty():
        return "Your cart is empty. Start shopping!"

    items_text = "\n".join([
```

```
        f"- {item.product.name} x{item.quantity}: ${item.subtotal}
            "
        for item in cart.items
    ])

    return f"""
    Your Cart ({cart.item_count} items):
    {items_text}

    Subtotal: ${cart.subtotal}
    """
```

15 Search Orchestration Pipeline

The search orchestration pipeline coordinates multiple retrieval strategies to find relevant products.

15.1 Pipeline Overview

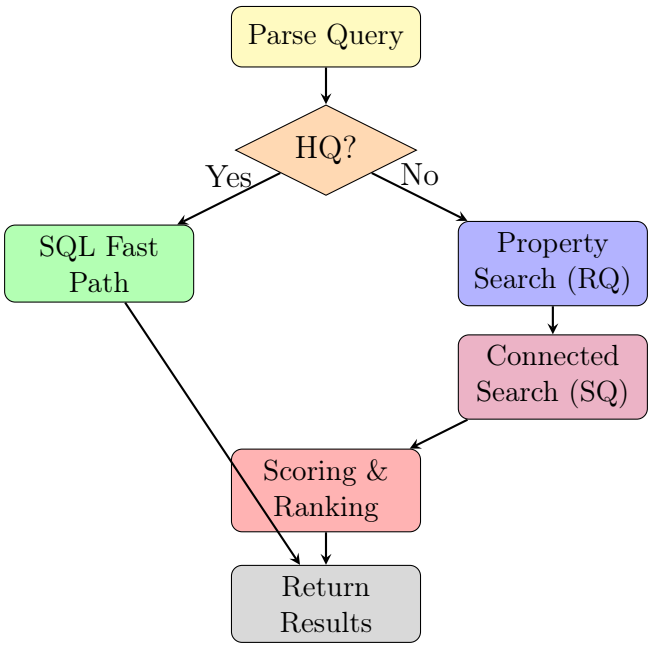

Figure 6: Search Orchestration Pipeline

15.2 HQ Fast Path

15.2.1 What

Direct SQL lookup bypassing semantic search.

15.2.2 When

- User indicates urgency

- Generic queries without specific requirements
- Re-queries after failed detailed search

15.2.3 Why

- **Speed**: Sub-100ms response time
- **Simplicity**: No embedding computation
- **Reliability**: Direct database access

Listing 32: HQ Fast Path Implementation

```
async def hq_fast_path(request: ProductRequest) -> List[Product]:
    """Execute fast SQL-based search."""

    query = """
    SELECT * FROM products
    WHERE category = :category
    AND status = 'active'
    ORDER BY popularity DESC
    LIMIT 20
    """

    return await sql_client.fetch_all(
        query,
        {"category": request.product_type}
    )
```

15.3 Property Search (RQ)

15.3.1 What

Dual-path semantic search using Main VDB and Property VDB.

15.3.2 When

- User specifies product properties (color, material, style)
- Fuzzy matching required
- Semantic understanding needed

15.3.3 Why

- Handles synonym matching ("crimson" → "red")
- Understands semantic similarity
- Enables fuzzy property matching

Listing 33: Property Search Implementation

```python
async def property_search(request: ProductRequest) -> List[
    ScoredProduct]:
    """Execute property-based semantic search."""

    results = []

    # Path 1: Main VDB search
    query_embedding = encoder.encode(request.to_search_string())
    main_results = await main_vdb.search(
        collection_name="products",
        query_vector=query_embedding,
        limit=50
    )

    # Path 2: Property VDB search
    for prop in request.properties:
        prop_embedding = encoder.encode(f"{prop.name}: {prop.value
            }")
        prop_results = await property_vdb.search(
            collection_name="product_properties",
            query_vector=prop_embedding,
            limit=30
        )

        # Weight by property importance
        for r in prop_results:
            r.score *= prop.weight

        results.extend(prop_results)

    # Merge and deduplicate
    return merge_results(main_results, results)
```

15.4 Connected Search (SQ)

15.4.1 What

Knowledge graph traversal for related products.

15.4.2 When

- User asks for recommendations
- "Something similar to..." queries
- Complementary product requests

15.4.3 Why

- Discovers non-obvious connections

- Leverages co-purchase patterns
- Enables style-based recommendations

Listing 34: Connected Search Implementation

```python
async def connected_search(
    seed_products: List[str],
    relation_types: List[str] = None
) -> List[Product]:
    """Find␣connected␣products␣via␣KG␣traversal."""

    if relation_types is None:
        relation_types = ["ALSO_BOUGHT", "SIMILAR_TO", "
           COMPLEMENTS"]

    query = """
␣␣␣␣MATCH␣(seed:Product)-[r]->(connected:Product)
␣␣␣␣WHERE␣seed.id␣IN␣$seed_ids
␣␣␣␣AND␣type(r)␣IN␣$relations
␣␣␣␣RETURN␣connected,␣type(r)␣as␣relation,␣r.strength␣as␣strength
␣␣␣␣ORDER␣BY␣r.strength␣DESC
␣␣␣␣LIMIT␣30
␣␣␣␣"""

    results = await memgraph.execute(query, {
        "seed_ids": seed_products,
        "relations": relation_types
    })

    return [
        ConnectedProduct(
            product=r["connected"],
            relation=r["relation"],
            strength=r["strength"]
        )
        for r in results
    ]
```

15.5 Subcategory Scoring

Products matching the user's intended subcategory receive a bonus:

Listing 35: Subcategory Scoring

```python
def apply_subcategory_bonus(
    products: List[ScoredProduct],
    target_subcategory: str
) -> List[ScoredProduct]:
    """Apply␣bonus␣for␣subcategory␣match."""

    target_embedding = encoder.encode(target_subcategory)
```

```
    for product in products:
        subcat_embedding = encoder.encode(product.subcategory)
        similarity = cosine_similarity(target_embedding,
           subcat_embedding)

        # Bonus scales with similarity
        bonus = similarity * SUBCATEGORY_BONUS_WEIGHT  # 0.2
        product.score += bonus

    return products
```

16 Weighting and Scoring System

16.1 Property Weight Formula

Individual property scores are calculated as:

$$score_{property} = similarity \times weight$$

Where:

- *similarity* is the cosine similarity from VDB search (0-1)
- *weight* is the user-assigned importance (0.5-2.0)

16.2 Score Combination Formula

The final product score combines multiple factors:

$$final\_score = \sum_i (property\_scores_i) + connected\_bonus + subcategory\_bonus - literal\_penalties$$

Listing 36: Score Combination Implementation

```
def calculate_final_score(
    product: Product,
    property_scores: Dict[str, float],
    connected_bonus: float,
    subcategory_bonus: float,
    literal_results: Dict[str, bool]
) -> float:
    """Calculate final product score."""

    # Sum property scores
    base_score = sum(property_scores.values())

    # Add bonuses
    total_score = base_score + connected_bonus + subcategory_bonus

```

```
    # Apply literal penalties
    for literal_name, passed in literal_results.items():
        if not passed:
            # Hard filter: product doesn't qualify
            return -1

    return total_score
```

16.3 Bonuses and Penalties

Factor	Value	Description
CONNECTED_BONUS	+0.15	Product found via KG traversal
SUBCATEGORY_BONUS	+0.20	Subcategory similarity bonus
EXACT_MATCH_BONUS	+0.25	Exact property value match
LITERAL_FAIL	Filter out	Product doesn't meet literal constraint

Table 6: Scoring Bonuses and Penalties

16.4 Superlative Re-ranking

For superlative queries, results are re-ranked with a 70/30 split:

Listing 37: Superlative Re-ranking

```
def rerank_for_superlative(
    products: List[ScoredProduct],
    sort_literal: str,
    ascending: bool
) -> List[ScoredProduct]:
    """Re-rank␣results␣for␣superlative␣queries."""

    # Sort by literal value
    sorted_by_literal = sorted(
        products,
        key=lambda p: p.product[sort_literal],
        reverse=not ascending
    )

    # Get rankings
    literal_ranks = {p.product.id: i for i, p in enumerate(
       sorted_by_literal)}
    semantic_ranks = {p.product.id: i for i, p in enumerate(
       products)}

    # Combine: 70% literal, 30% semantic
```

```
    for product in products:
        pid = product.product.id
        combined_rank = 0.7 * literal_ranks[pid] + 0.3 *
            semantic_ranks[pid]
        product.final_rank = combined_rank

    return sorted(products, key=lambda p: p.final_rank)
```

17 User Context Enrichment

BeeKurse maintains user context to personalize search results across conversation turns.

17.1 Three-Stage Pipeline

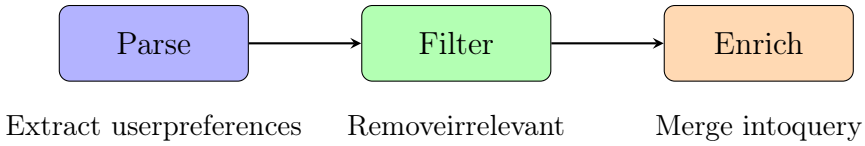

Figure 7: User Context Pipeline

17.2 User Preferences

Listing 38: User Context Model

```
class UserContext(BaseModel):
    user_id: str
    gender: Optional[str] = None
    size_preferences: Dict[str, str] = {}  # category -> size
    age_group: Optional[str] = None
    style_preferences: List[str] = []
    price_range: Optional[Tuple[float, float]] = None
    brand_preferences: List[str] = []
    recent_searches: List[str] = []
    viewed_products: List[str] = []
    purchased_products: List[str] = []
```

17.3 Property Merging Strategy

Listing 39: Context Merging

```
def merge_user_context(
    request: ProductRequest,
    context: UserContext
) -> ProductRequest:
    """Merge user context into product request."""

```

```
    enriched = request.copy()

    # Add gender if relevant and not specified
    if context.gender and not has_property(request, "gender"):
        if is_gendered_category(request.product_type):
            enriched.properties.append(
                Property(name="gender", value=context.gender,
                    weight=0.5)
            )

    # Add size preference
    if request.product_type in context.size_preferences:
        if not has_property(request, "size"):
            enriched.properties.append(
                Property(
                    name="size",
                    value=context.size_preferences[request.
                        product_type],
                    weight=0.5
                )
            )

    # Add style preferences with low weight
    for style in context.style_preferences[:2]:  # Max 2
        if not has_property(request, "style"):
            enriched.properties.append(
                Property(name="style", value=style, weight=0.3)
            )

    return enriched
```

18 Conclusion

18.1 Summary

This report has presented KURSE, a neuro-symbolic inference engine combining neural language understanding with symbolic knowledge graph reasoning. Key contributions include:

1. **Tri-Store Architecture**: Integration of SQL, vector, and graph databases for comprehensive data management
2. **N-ary Event Representation**: Novel approach to capturing complex multi-participant events beyond binary relations
3. **Two-Phase KG Ingestion**: Streaming pipeline with event identification and relation extraction phases
4. **Ontology-Driven Reasoning**: Formal axioms enabling inference through transitivity, symmetry, and constraint validation

5. **BeeKurse Application**: Demonstration of KURSE adaptability for conversational e-commerce with:
    - Sophisticated query classification (5 types)
    - Multi-strategy search orchestration
    - Intelligent scoring and re-ranking
    - User context personalization