Comprehensive Implementation Guide: Argumentation Schemes

Research Paper Analysis & Development Reference

Executive Summary

This document synthesizes the research paper "Argumentation Schemes: History, Classifications, and Computational Applications" (Macagno, Walton, Reed, 2017) into an actionable reference for software development teams implementing argumentation analysis systems.

Core Purpose: Argumentation schemes are abstract structures representing the most generic types of argument used in everyday reasoning. They serve as building blocks for analyzing, evaluating, and constructing arguments computationally.

Primary Use Cases:

- Argument analysis and identification in text
- Automated argument mining
- AI-powered debate systems
- Legal reasoning systems
- Educational critical thinking tools
- Dialogue and decision-making systems

1. Fundamental Concepts

1.1 What Are Argumentation Schemes?

Definition: Argumentation schemes are stereotypical patterns of inference that combine semantic-ontological relations with types of reasoning and logical axioms. They represent the abstract structure of common natural arguments.

Key Characteristics:

- **Defeasible**: Can be defeated by counterarguments or critical questions
- **Premise-Conclusion Structure**: Abstract form with variables and constants
- **Critical Questions**: Each scheme has corresponding questions representing defeasibility conditions
- **Modular**: Can be combined in networks to represent complex arguments

1.2 Core Components

Every argumentation scheme consists of:

1. **Premises**: The starting points or evidence (may be explicit or implicit)
2. **Conclusion**: The claim being supported
3. **Warrant/Inference Rule**: The connection justifying the move from premises to conclusion
4. **Critical Questions**: Structured challenges that can defeat the argument

Example - Argument from Expert Opinion:

```
Premise 1: Source E is an expert in domain D
Premise 2: E asserts that proposition A is true
Premise 3: A is within domain D
Conclusion: A may plausibly be taken to be true

Critical Questions:
- How credible is E as an expert source?
- Is E reliable?
- Is A consistent with what other experts say?
- Is E's assertion based on evidence?
```

2. Historical Foundation & Evolution

2.1 Ancient Origins (Aristotle, Cicero, Boethius)

Aristotelian Topics:

- Original concept: "places to find arguments"
- **Topoi** as conditional principles: "If P, then Q"
- Function: Generic principles from which specific premises can be instantiated
- Distinction between **generic topics** (abstract, universal) and **specific topics** (domain-specific)

Cicero's Classification:

- **Intrinsic arguments**: Based on subject matter properties (definition, cause, classification)
- **Extrinsic arguments**: Based on external authority or source
- Connected to **stasis theory**: Organizing arguments by issue type (conjecture, definition, qualification)

Boethius's Systematization:

- Introduced **maximae propositiones** (general axioms) and **differentiae** (genera of maxims)
- Three categories:
    - Intrinsic loci (from substance)

- Intermediate loci (from grammatical/semantic relations)
- Extrinsic loci (from external factors)

Implementation Note: Historical classification provides theoretical foundation but modern systems use updated taxonomies (see Section 3).

3. Modern Classification Systems

3.1 Purpose-Based Classification (Primary Recommendation)

The paper proposes a **pragmatic classification** based on the purpose of arguments:

```
Purpose of Argument
├── Assess desirability of action (Practical)
│   ├── Internal (quality of action itself)
│   │   ├── Means to achieve goal
│   │   │   └── Schemes: Practical reasoning, value-based
practical reasoning
│   │   └── Consequences of action
│   │       └── Schemes: Argument from consequences,
slippery slope
│   └── External (authority of source)
│       ├── Authority
│       │   └── Schemes: Argument from position to know
│       └── Popular practice
│           └── Schemes: Argument ad populum
│
└── Establish acceptability of judgment (Epistemic)
    ├── Internal (properties of subject matter)
    │   ├── Predict/retrodict events
    │   │   └── Schemes: Argument from cause to effect,
argument from sign
    │   ├── Classify entities (factual judgment)
    │   │   └── Schemes: Argument from classification,
definition, genus-species
    │   └── Attribute value (evaluative judgment)
    │       └── Schemes: Argument from values, verbal
classification
    └── External (source quality)
        ├── Expert knowledge
        │   └── Schemes: Argument from expert opinion
        │
```

```
    └── Reliability of source
        └── Schemes: Ad hominem (negative)
```

Implementation Priority: This hierarchical structure should drive UI organization and scheme selection algorithms.

3.2 Alternative Classification Systems

Perelman & Olbrechts-Tyteca (New Rhetoric):

- Association vs. Dissociation
- Quasi-logical, relations establishing reality structure, arguments based on reality structure
- *Limitation*: Multiple inconsistent criteria

Toulmin:

- 9 classes based on warrant function
- Mixed criteria (reasoning type, logical rules, content)
- *Limitation*: No clear hierarchy

Kienpointner (Alltagslogik):

- Based on: (1) inference type, (2) epistemic nature, (3) dialectical function, (4) pragmatic function
- Three abstract classes: using rules, establishing rules, using+establishing rules
- *Strength*: Separates reasoning type from content
- *Limitation*: Complex for inductive schemes

Pragma-Dialectics (Van Eemeren):

- Three basic schemes: symptomatic, similarity-based, instrumental
- *Limitation*: Mixes material relations with reasoning types

Implementation Recommendation: Use purpose-based classification (3.1) as primary taxonomy, with Kienpointner's inference-type distinction as secondary attribute.

4. Critical Schemes for Implementation

4.1 Priority Tier 1: Most Common Schemes

Based on empirical analysis (Feng & Hirst, 2011), these five schemes constitute **61% of natural arguments**:

4.1.1 Argument from Example (Inductive)

```
Premise: In case C, which is an example of generalization
G, property P holds
Conclusion: Therefore, generally, in cases of type G,
property P holds
```

```
Critical Questions:
- Is the example representative?
- Are there counterexamples?
- Is the sample size sufficient?
- Are there relevant differences between the example and
the general case?
```

4.1.2 Argument from Cause to Effect (Deductive/Abductive)

```
Major Premise: Generally, if A occurs, then B will (might)
occur
Minor Premise: In this case, A occurs (might occur)
Conclusion: Therefore, in this case, B will (might) occur

Critical Questions:
- How strong is the causal relationship?
- Are there intervening factors that could prevent the
effect?
- Is the cause necessary, sufficient, or contributory?
- Are there alternative causes?
```

Variations (implement as separate schemes):

- **Effect to Cause** (abductive): B occurred, therefore A might have caused it
- **Cause to No Effect** (modus tollens): B didn't occur, therefore A didn't occur
- **No Cause to No Effect** (contraposition + abduction)

4.1.3 Practical Reasoning (Practical)

```
Premise 1: Agent has goal G
Premise 2: Carrying out action A is a means to realize G
Conclusion: Therefore, agent should carry out action A

Critical Questions:
- What other goals might conflict with G?
- What alternative actions could achieve G?
- Which action is most efficient?
- Is it practically possible to perform A?
- What are the side effects/consequences of A?
```

Important: This is the foundation for value-based practical reasoning (see 4.2.3).

4.1.4 Argument from Consequences (Practical-Evaluative)

Positive Version:

```
Premise: If A is brought about, good consequences will
plausibly occur
Conclusion: Therefore A should be brought about
```

Negative Version:

```
Premise: If A is brought about, bad consequences will
plausibly occur
Conclusion: Therefore A should not be brought about
```

Critical Questions:

- How probable are the consequences?
- How significant are they?
- What other consequences should be considered?
- Are there ways to mitigate negative consequences?

4.1.5 Argument from Verbal Classification (Definitional)

```
Premise 1: Individual a has property F
Premise 2: For all x, if x has property F, then x is
classified as G
Conclusion: Individual a is classified as G

Critical Questions:
- Does a really have property F?
- Is F sufficient for classification as G?
- Is the definition of G accurate/accepted?
- Are there exceptions to the classification rule?
```

4.2 Priority Tier 2: Decision-Making Cluster

These schemes form an interconnected cluster essential for practical reasoning systems:

4.2.1 Value-Based Practical Reasoning (Complex)

```
Premise 1: Agent has goal G
Premise 2: G is supported by agent's values V
Premise 3: Bringing about A is necessary/sufficient to
bring about G
Conclusion: Therefore, agent should bring about A

Critical Questions:
- Same as practical reasoning, plus:
- What values support this goal?
- Are there conflicting values?
```

- How should value conflicts be resolved?

Relationship: This scheme **combines** basic practical reasoning with argument from values.

4.2.2 Argument from Values

Positive Version:

Premise 1: Value V is positive as judged by agent A
Premise 2: If V is positive, it is a reason for A to commit to goal G
Conclusion: V is a reason for A to commit to goal G

Negative Version:

Premise 1: Value V is negative as judged by agent A
Premise 2: If V is negative, it is a reason for retracting commitment to goal G
Conclusion: V is a reason for retracting commitment to goal G

4.2.3 Slippery Slope Argument (Complex Warning)

First Step Premise: A_0 is under consideration and seems acceptable
Recursive Premise: A_0 would lead to A_1, which leads to A_2, ..., which leads to A_n
Bad Outcome Premise: A_n is a horrible/disastrous outcome
Conclusion: Therefore A_0 should not be brought about

Critical Questions:
- What are the intervening steps between A_0 and A_n?
- How plausible is each transition?
- What are the weakest links in the chain?
- Can the chain be broken at any point?

Subtypes (implement as variations):

- Causal slippery slope
- Precedent slippery slope
- Linguistic/vagueness slippery slope

Cluster Relationships:

```
Instrumental Practical Reasoning
        ↓ (adds values)
Value-Based Practical Reasoning
```

```
        ↓ (foundation for)
Basic Slippery Slope ← (species of) ← Argument from
Negative Consequences
                                              ↑ (uses)
                                      Argument from Values
```

4.3 Priority Tier 3: Epistemic Schemes

4.3.1 Argument from Expert Opinion

```
Premise 1: Source E is an expert in domain D
Premise 2: E asserts that proposition A is true
Premise 3: A is within domain D
Conclusion: A may plausibly be taken to be true

Critical Questions:
- How credible is E as an expert?
- Is E reliable/trustworthy?
- Is A consistent with what other experts assert?
- Is E's assertion based on evidence?
```

Implementation Note: High importance for credibility assessment systems and fact-checking applications.

4.3.2 Argument from Sign (Abductive)

```
Premise: Event/property B is observed
Premise: B is a sign/symptom of event/property A
Conclusion: Therefore A (probably) occurred/exists

Critical Questions:
- How reliable is B as a sign of A?
- Are there alternative explanations for B?
- Is B a necessary or sufficient sign?
- What is the strength of the correlation?
```

4.3.3 Argument from Analogy (Analogical)

```
Premise 1: Case C1 has properties P1, P2, ..., Pn
Premise 2: Case C2 has properties P1, P2, ..., Pn
Premise 3: C1 also has property Q
Conclusion: Therefore C2 also has property Q

Critical Questions:
```

- How similar are C_1 and C_2?
- Are the shared properties relevant to Q?
- Are there relevant differences?
- How many analogous cases support the conclusion?

4.4 Additional Important Schemes

Argument from Commitment

Premise: Agent A is committed to proposition P
Premise: P implies/is inconsistent with proposition Q
Conclusion: Therefore A is (or should be) committed to Q / should retract commitment to P

Critical Questions:
- Is A really committed to P?
- Does P really imply Q?
- Could A give up commitment to P instead?

Argument from Position to Know

Premise: Agent A is in a position to know about things in domain D
Premise: A asserts that P is true (or false)
Premise: P is in domain D
Conclusion: P may plausibly be taken to be true (or false)

Critical Questions:
- Is A really in a position to know?
- Is A honest/sincere?
- Did A make an assertion about P?

Ad Hominem Argument (Attack on Credibility)

Premise: Agent A has character defect C or circumstantial bias B
Premise: C or B affects A's credibility regarding claim P
Conclusion: Therefore P is less credible / A's argument should be given less weight

Critical Questions:
- Does A really have defect C or bias B?
- Is it relevant to the issue at hand?

```
- Does it actually affect credibility on this claim?
```

5. Argument Structure: Networks and Modularity

5.1 The Network Principle

Critical Concept: Real arguments are rarely captured by a single scheme. Natural argumentation consists of **nets of interconnected schemes**.

Example from the paper (William Hague speech on Russia-Ukraine):

```
"Be in no doubt, there will be consequences. The world cannot say it is OK
to violate the sovereignty of other nations. This clearly is a violation of
the sovereignty independence and territorial integrity of Ukraine."
```

Scheme Network:

1. **Classification**: Russia's action → "violation of sovereignty" (Argument from Verbal Classification)
2. **Values/Commitment**: World's values → commitment against violations (Argument from Values + Commitment)
3. **Consequences**: Commitment → consequences against Russia (Argument from Consequences)

Network Structure:

```
Factual Observation
        ↓
[Verbal Classification]
        ↓
Classified State of Affairs
        ↓
[Argument from Values]
        ↓
Commitment to Response
        ↓
[Argument from Consequences]
        ↓
Action Recommendation
```

5.2 Implementation Strategy for Networks

Data Structure Requirements:

```
interface ArgumentNode {
  id: string;
  type: 'premise' | 'conclusion' | 'scheme';
  content: string;
  explicit: boolean; // Was it stated or implicit?
}

interface ArgumentScheme {
  id: string;
  schemeType: SchemeType;
  premises: ArgumentNode[];
  conclusion: ArgumentNode;
  criticalQuestions: CriticalQuestion[];
}

interface ArgumentNetwork {
  nodes: ArgumentNode[];
  schemes: ArgumentScheme[];
  edges: {
    from: string;
    to: string;
    relationship: 'supports' | 'attacks' | 'presupposes';
  }[];
}
```

Traversal Algorithm Needs:

- Forward chaining: From facts to conclusions
- Backward chaining: From desired conclusion to required premises
- Bidirectional: Meet-in-the-middle for complex arguments

5.3 Enthymeme Reconstruction

Definition: Enthymemes are arguments with implicit premises or conclusions.

Implementation Challenge: Systems must infer unstated components.

Approaches:

1. **Schema-Driven**: If scheme is identified, missing components can be inferred from scheme structure

    - Example: If we detect "X is an expert" and "X says P", we can infer the scheme is Expert Opinion and add implicit premise "P is in X's domain"
2. **Semantic Similarity**: Use embeddings to find implicit connections
    - Bridge concepts between stated premises and conclusion
3. **Common Knowledge**: Leverage knowledge bases for standard assumptions
    - Example: "He had fever, so he was breathing fast" assumes "fever causes fast breathing"

Critical: The paper's analysis of Abelard shows how **assumptions** connect the major premise (maxim) to the specific case.

6. Critical Questions: Types and Handling

6.1 Four Functional Types (Verheij Classification)

Understanding how critical questions work is **essential** for proper implementation:

Type 1: Premise Attack

Function: Questions whether a premise actually holds **Example**: "Is E really an expert in field F?" **Implementation**:

- Creates counterargument targeting specific premise
- Shifts burden of proof back to original arguer
- If premise is defeated, argument collapses

Type 2: Exception/Undercutting

Function: Points to exceptional situations where scheme defaults **Example**: "Is there evidence that E is biased?" **Implementation**:

- Creates undercutting defeater
- Doesn't deny premises or conclusion, but breaks inferential link
- Often captured as additional premise (exception condition)

Type 3: Usage Conditions

Function: Frames conditions for proper use of scheme **Example**: "Is the source credible?" **Implementation**:

- Pre-conditions that must be met
- Can be modeled as presumptions (assumed unless challenged)

Type 4: Counterargument Pointer

Function: Indicates other arguments that might attack **Example**: "What do other experts say?" **Implementation**:

- Triggers search for conflicting arguments
- May not defeat on its own, requires evidence

6.2 Burden of Proof Allocation

Critical Design Decision: How does asking a critical question affect burden of proof?

Three Premise Types (Gordon & Walton model in Carneades):

1. **Ordinary Premises**:
   - Must be supported by arguments even if unquestioned
   - Default: Burden on proponent
2. **Assumptions**:
   - Accepted unless questioned
   - Once questioned, burden shifts to proponent to defend
3. **Exceptions**:
   - Assumed not to apply unless questioner provides evidence
   - Burden on questioner to show exception applies

Implementation Example:

```
enum PremiseType {
  ORDINARY,     // Needs support
  ASSUMPTION,   // Accepted until questioned
  EXCEPTION     // Questioner must prove
}

interface Premise {
  content: string;
  type: PremiseType;
  supported: boolean;
  questioned: boolean;
}

function evaluatePremise(premise: Premise): boolean {
  if (premise.type === PremiseType.ORDINARY) {
    return premise.supported;
  } else if (premise.type === PremiseType.ASSUMPTION) {
    return !premise.questioned || premise.supported;
  } else { // EXCEPTION
```

```
    return !(premise.questioned && premise.supported);
  }
}
```

6.3 Critical Question Matching

Each scheme must have **standardized critical questions**.

Example Template (Argument from Expert Opinion):

```
const expertOpinionCQs: CriticalQuestion[] = [
  {
    id: 'eo_cq1',
    text: 'How credible is E as an expert source?',
    type: 'USAGE_CONDITION',
    premiseType: 'ASSUMPTION',
    targets: ['expert_credibility']
  },
  {
    id: 'eo_cq2',
    text: 'Is E an expert in the field F that A is in?',
    type: 'PREMISE_ATTACK',
    premiseType: 'ORDINARY',
    targets: ['expertise_premise']
  },
  {
    id: 'eo_cq3',
    text: 'What did E assert that implies A?',
    type: 'PREMISE_ATTACK',
    premiseType: 'ORDINARY',
    targets: ['assertion_premise']
  },
  {
    id: 'eo_cq4',
    text: 'Is E personally reliable as a source?',
    type: 'EXCEPTION',
    premiseType: 'EXCEPTION',
    targets: ['reliability']
  },
  {
    id: 'eo_cq5',
```

```
    text: 'Is A consistent with what other experts
assert?',
    type: 'COUNTERARGUMENT',
    premiseType: 'EXCEPTION',
    targets: ['expert_consensus']
  },
  {
    id: 'eo_cq6',
    text: 'Is E\'s assertion based on evidence?',
    type: 'USAGE_CONDITION',
    premiseType: 'ASSUMPTION',
    targets: ['evidential_basis']
  }
];
```

7. Formal Representation & Computational Models

7.1 Abstract Argumentation Frameworks (Dung 1995)

Foundation: Arguments and attack relations form a directed graph.

Basic Structure:

```
Args: Set of arguments {a₁, a₂, ..., aₙ}
R: Attack relation (aᵢ attacks aⱼ)

Semantics:
- An argument is "in" (accepted) if all its attackers are
"out"
- An argument is "out" (rejected) if it's attacked by an
"in" argument
- An argument is "undecided" otherwise
```

Implementation: Foundation for many systems, but abstract (no internal argument structure).

7.2 ASPIC+ Framework

Extension of Dung: Adds structured arguments with:

- Strict rules (deductive)
- Defeasible rules (schemes)
- Preferences among arguments

Key Features:

```
Arguments have structure:
- Set of premises
- Sequence of inferences
- Conclusion

Attack types:
- Undermining: attacks premise
- Rebutting: attacks conclusion
- Undercutting: attacks inference
```

Use Case: Legal reasoning, case-based reasoning (see Section 8.2).

7.3 Carneades Argumentation System

Current State: Version 4 (CAS2 model), actively maintained.

Key Innovation: Handles scheme representation directly.

Scheme Definition in CAS2:

```
Scheme = (e, v, g) where:
- e: weighing function (evaluates argument strength)
- v: validation function (tests if argument instantiates
scheme)
- g: generation function (creates arguments from scheme)
```

Argument Structure:

```
Argument = (S, P, C, U) where:
- S: scheme instantiated
- P: set of premises
- C: conclusion
- U: undercutter (optional)
```

Tripartite Graphs:

- Statement nodes (propositions)
- Argument nodes (inferences)
- Issue nodes (questions under dispute)

Evaluation:

- Label statements: in, out, undecided
- Apply proof standards
- Consider cumulative arguments

Implementation Advantages:

- Supports all major schemes (20+ built-in)

- Handles forwards and backwards reasoning
- Open source: https://github.com/carneades
- Web interface: http://carneades.fokus.fraunhofer.de/carneades

Recommendation: Strong candidate for production systems requiring full scheme support.

7.4 Argument Interchange Format (AIF)

Purpose: Standard representation for argument interchange.

Core Ontology: Description logic specification.

Key Components:

- I-nodes: Information nodes (propositions)
- S-nodes: Scheme nodes (inference rules)
- RA-nodes: Rule application (instantiated inferences)

Extensions:

- AIF+ ontology: Adds dialogue structure
- Scheme ontology: Formal specification of all schemes

Tools Using AIF:

- OVA (Online Visualization of Argument)
- AIFdb (database of analyzed arguments)
- Araucaria (legacy analysis tool)

Benefit: Interoperability between different argument tools and systems.

8. Domain-Specific Applications

8.1 AI & Law: Case-Based Reasoning

Context: Legal arguments depend heavily on precedent cases.

Special Schemes for Legal Reasoning:

Scheme CS1 (Prakken et al. 2015):

```
commonPfactors(current, precedent) = p
commonDfactors(current, precedent) = d
preferred(p, d) in precedent
─────────────────────────────────────
outcome(current) = Plaintiff

Where:
```

```
- Pfactors: factors favoring plaintiff
- Dfactors: factors favoring defendant
- preferred: preference relation established in precedent
```

Factors: Abstract features shared between cases

- Example: "contract was breached", "defendant acted in bad faith"

Attack Patterns:

- Distinguish: show relevant differences
- Counter-example: cite precedent with same factors, different outcome
- Reinterpret: change factor characterization

Implementation Requirements:

- Factor ontology for domain
- Case database with factor annotations
- Preference ordering system
- Similarity metrics

8.2 Statutory Interpretation

Context: Interpreting legal statutes requires specialized argument forms.

Key Schemes (Walton, Sartor, Macagno 2016):

Argument from Ordinary Meaning:

```
Premise: Text T contains term W
Premise: W has ordinary meaning M
Conclusion: T should be interpreted according to M
```

Argument from Technical Meaning:

```
Premise: Text T is in specialized domain D
Premise: In D, term W has technical meaning M_t
Conclusion: T should be interpreted according to M_t
```

Argument from Purpose:

```
Premise: Statute S was enacted to achieve purpose P
Premise: Interpretation I of S best serves P
Conclusion: S should be interpreted as I
```

A Contrario Argument:

```
Premise: Statute S explicitly applies to cases of type C
Premise: Case X is not of type C
Conclusion: S does not apply to X
```

Implementation: These can be integrated into legal AI systems using CAS or ASPIC+.

8.3 Science Education

Use Case: Teaching students to argue scientifically and evaluate evidence.

Applications:

- Representing student arguments
- Identifying argumentation quality
- Reconstructing implicit premises
- Assessing reasoning systematically

Key Schemes in Science:

- Argument from example/experiment
- Argument from cause to effect
- Argument from sign (observation → hypothesis)
- Argument from expert opinion (citing research)

Challenges:

- Students conflate different schemes
- Need clear differentiation criteria
- Must teach both construction and evaluation

Implementation Guidance: Classification system (Section 3.1) addresses differentiation problem.

9. Argument Mining & Natural Language Processing

9.1 The Challenge

Task: Automatically extract argument structure from natural language text.

Difficulty Factors:

1. **Data scarcity**: Analyzing argument structure is time-consuming
2. **Complexity**: Schemes have thousands of possible combinations
3. **Implicitness**: Much is left unstated (enthymemes)
4. **Context-dependence**: Same words, different schemes

9.2 Available Datasets

AIFdb Corpora

- **Location**: corpora.aifdb.org
- **Format**: AIF (Argument Interchange Format)
- **Content**: Analyzed argumentation with scheme annotations
- **Tool**: OVA for creating analyses

Key Corpora:

- AraucariaDB (original, legacy)
- Moral Maze excerpts (35 scheme instances)
- ExpertOpinion-PositiveConsequences (71 examples)

Internet Argument Corpus (IAC)

- **Size**: 390,000 quote-response pairs
- **Limitation**: Thin conception of argument (polarity-focused)
- **Use**: Sentiment analysis, basic stance detection

Potsdam Microtext Corpus

- **Size**: 130 parallel English/German arguments
- **Structure**: Freeman-style (linked/convergent, rebutting/undercutting)
- **Constraint**: All arguments have exactly 5 components
- **Use**: Controlled testing environment

Recommendation: Use AIFdb for scheme-specific training; Microtext for structural testing.

9.3 Successful Approaches

Feng & Hirst (2011): Scheme Classification

Method: Machine learning to classify arguments into scheme types.

Features Used:

- Keywords/phrases (e.g., "should", "must", "need" for practical reasoning)
- Part-of-speech patterns
- Semantic features
- Discourse markers

Results: Classification accuracy 0.64 to 0.98 for 5 most common schemes.

Implementation: 28 keywords identified for practical reasoning alone.

Practical Application:

```
practical_reasoning_indicators = [
    'want', 'aim', 'objective', 'goal', 'should', 'must',
'need',
    'ought', 'have to', 'required', 'necessary', 'in order
to',
    'so that', 'to achieve', 'purpose', 'intend', ...
]

expert_opinion_indicators = [
```

```
    'expert', 'said', 'stated', 'according to', 'claims',
'asserts',
    'specialist', 'authority', 'researcher', 'studies
show', ...
]
```

Lawrence & Reed (2015): Scheme-Guided Structure Detection

Key Insight: Schemes constrain proposition types, which helps identify structure.

Method:

1. Identify proposition types (normative, factual, attributive)
2. Match proposition type patterns to schemes
3. Use scheme structure to guide argument reconstruction

Example:

- Detect "said" → likely reported speech → check for expert opinion scheme
- Find "expert" nearby → strengthen hypothesis
- Look for proposition with semantic similarity to quote → likely conclusion

Results: F1 of 0.59-0.91 for component detection, 0.62-0.88 for scheme instances.

Implementation Requirement: Scheme ontology specifying:

- Which proposition types appear as premises/conclusions
- Lexical triggers for each scheme
- Semantic relationships between components

9.4 Implementation Strategy for Mining

Pipeline Architecture:

```
1. Preprocessing
   ├─ Sentence segmentation
   ├─ POS tagging
   ├─ Dependency parsing
   └─ Named entity recognition

2. Component Detection
   ├─ Argumentative vs. non-argumentative
   ├─ Premise vs. conclusion identification
   └─ Boundary detection

3. Proposition Type Classification
   ├─ Factual/normative/evaluative
```

```
    ├─ Speech acts (assertion, question, command)
    └─ Semantic roles

4. Scheme Recognition
    ├─ Keyword/pattern matching
    ├─ Type-based filtering
    └─ ML classification

5. Structure Assembly
    ├─ Link premises to conclusions
    ├─ Identify implicit components
    └─ Build argument graph

6. Scheme Validation
    ├─ Check component types match
    ├─ Verify semantic coherence
    └─ Apply critical questions
```

Machine Learning Features:

```
features = {
    'lexical': ['keywords', 'ngrams', 'pos_patterns'],
    'semantic': ['word_embeddings', 'semantic_roles',
'sentiment'],
    'discourse': ['connectives', 'discourse_relations',
'position'],
    'syntactic': ['dependency_patterns',
'clause_structure'],
    'scheme_specific': ['proposition_types', 'modal_verbs',
'speech_verbs']
}
```

10. Ontology Engineering

10.1 Argument Scheme Ontology

Purpose: Formal, machine-readable specification of schemes for automated reasoning.

Format: OWL (Web Ontology Language) / Description Logic

Location: http://arg.tech/aif.owl

Core Components:

Class Hierarchy:

```
RA (Rule Application)
├─ Inference (general)
│   ├─ DeductiveInference
│   ├─ InductiveInference
│   └─ AbductiveInference
├─ Scheme (specific types)
│   ├─ ExpertOpinionInference
│   ├─ CauseToEffectInference
│   ├─ PracticalReasoningInference
│   └─ [60+ specific schemes]
```

Properties:

```
hasConclusion: links scheme to conclusion node
hasPremise: links scheme to premise nodes
  - hasPresumption: premise assumed unless challenged
  - hasException: premise assumed false unless proven
hasCriticalQuestion: links to standard questions
```

Example Snippet (Expert Opinion):

```
<Class IRI="#ExpertOpinion_Inference"/>
  <ObjectIntersectionOf>
    <Class IRI="#Presumptive_Inference"/>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasConclusion"/>
      <Class IRI="#KnowledgePosition_Statement"/>
    </ObjectSomeValuesFrom>
    <ObjectSomeValuesFrom>
      <ObjectProperty IRI="#hasFieldExpertise_Premise"/>
      <Class IRI="#FieldExpertise_Statement"/>
    </ObjectSomeValuesFrom>
    ...
  </ObjectIntersectionOf>
</Class>
```

10.2 Benefits of Formal Ontology

1. Automated Reasoning:

- **Transitivity**: If A supports B, and B supports C, infer A supports C

- **Classification**: Automatically place schemes in taxonomy
- **Subsumption**: Fear appeal is automatically classified as negative consequences

2. Critical Question Inheritance:

```
If SchemeA is a subclass of SchemeB,
then all CQs of SchemeB apply to SchemeA.

Example:
- Fear appeal is subclass of negative consequences
- All negative consequence CQs automatically apply to fear
appeal
- Plus fear appeal adds specific CQs about emotion
manipulation
```

3. Consistency Checking:

- Detect contradictions in scheme definitions
- Verify all required components are specified
- Ensure critical questions target actual components

4. Search and Retrieval:

- Query for all causal schemes
- Find schemes with specific premise types
- Locate schemes applicable to given propositions

10.3 Implementation Guidelines

For Developers:

1. **Use Existing Ontology**: Don't reinvent; extend http://arg.tech/aif.owl

2. **Reasoning Engine**: Integrate OWL reasoner (e.g., Pellet, HermiT, ELK)

3. **Scheme Instantiation**:

```
class SchemeInstance {
  schemeType: SchemeClass; // from ontology
  premises: Map<PremiseRole, Proposition>;
  conclusion: Proposition;

  validate(): boolean {
    // Use reasoner to check if instance matches scheme
definition
    return this.schemeType.checkConsistency(this);
  }
```

```
  getCriticalQuestions(): CriticalQuestion[] {
    // Retrieve from ontology including inherited CQs
    return this.schemeType.getAllCriticalQuestions();
  }
}
```

4. **SPARQL Queries for Scheme Selection**:

```
# Find all schemes with conclusions about actions
SELECT ?scheme
WHERE {
  ?scheme rdfs:subClassOf* :Inference .
  ?scheme :hasConclusion ?conclusion .
  ?conclusion rdf:type :Action_Statement .
}
```

11. System Architecture Recommendations

11.1 Core Modules

```
┌─────────────────────────────────────────────────────┐
│            User Interface Layer                     │
│  ┌──────────────┐  ┌──────────────┐  ┌────────────┐ │
│  │ Argument     │  │   Critical   │  │  Argument  │ │
│  │ Constructor  │  │   Question   │  │   Map      │ │
│  │              │  │   Generator  │  │  Visualizer│ │
│  └──────────────┘  └──────────────┘  └────────────┘ │
└─────────────────────────────────────────────────────┘
                          ↓
┌─────────────────────────────────────────────────────┐
│         Argumentation Engine                        │
│  ┌──────────────┐  ┌──────────────┐  ┌────────────┐ │
│  │   Scheme     │  │  Argument    │  │  Argument  │ │
│  │  Recognizer  │  │  Evaluator   │  │  Generator │ │
│  │              │  │              │  │            │ │
│  └──────────────┘  └──────────────┘  └────────────┘ │
└─────────────────────────────────────────────────────┘
                          ↓
┌─────────────────────────────────────────────────────┐
│           Knowledge Layer                           │
│  ┌──────────────┐  ┌──────────────┐  ┌────────────┐ │
│  │   Scheme     │  │   Argument   │  │   Domain   │ │
```

```
│  │    Ontology    │  │    Database    │  │  Knowledge   │ │
│  │                │  │    (AIFdb)     │  │    Base      │ │
│  └────────────────┘  └────────────────┘  └──────────────┘ │
└───────────────────────────────────────────────────────────┘
                              ↓
┌───────────────────────────────────────────────────────────┐
│       Infrastructure Services                             │
│  ┌────────────────┐  ┌────────────────┐  ┌──────────────┐ │
│  │     NLP        │  │  OWL Reasoner  │  │   Formal     │ │
│  │    Pipeline    │  │  (Pellet/      │  │  Framework   │ │
│  │                │  │   HermiT)      │  │  (Carneades) │ │
│  └────────────────┘  └────────────────┘  └──────────────┘ │
└───────────────────────────────────────────────────────────┘
```

11.2 Scheme Recognizer Module

Responsibilities:

1. Analyze text to identify potential arguments
2. Classify arguments by scheme type
3. Extract scheme components (premises, conclusion)
4. Handle implicit components (enthymeme reconstruction)

Algorithm Outline:

```
class SchemeRecognizer:
    def __init__(self, ontology, ml_model, keyword_db):
        self.ontology = ontology
        self.classifier = ml_model
        self.keywords = keyword_db

    def recognize(self, text: str) -> List[SchemeInstance]:
        # 1. Preprocessing
        sentences = segment_sentences(text)
        parse_trees = dependency_parse(sentences)

        # 2. Component detection
        components =
self.detect_argumentative_components(sentences)

        # 3. Proposition typing
        for comp in components:
```

```
            comp.prop_type =
self.classify_proposition_type(comp)

        # 4. Scheme hypothesis generation
        hypotheses = []
        for pattern in self.keyword_patterns:
            if pattern.matches(text):

hypotheses.extend(pattern.suggested_schemes)

        # 5. ML-based classification
        for hyp in hypotheses:
            hyp.confidence = self.classifier.predict_proba(
                extract_features(components, hyp)
            )

        # 6. Ontology-based validation
        valid_hypotheses = [
            h for h in hypotheses
            if self.ontology.validate_scheme_instance(h)
        ]

        # 7. Structure assembly
        schemes = self.assemble_argument_structures(
            valid_hypotheses, components
        )

        # 8. Enthymeme reconstruction
        for scheme in schemes:
            scheme.fill_implicit_components(self.ontology)

        return schemes
```

11.3 Argument Evaluator Module

Responsibilities:

1. Generate critical questions for identified schemes
2. Check for defeaters (attacks)
3. Compute acceptability status
4. Apply proof standards

Algorithm Outline:

```
class ArgumentEvaluator:
    def __init__(self, ontology, knowledge_base):
        self.ontology = ontology
        self.kb = knowledge_base

    def evaluate(self, arg_graph: ArgumentGraph) ->
EvaluationResult:
        # 1. Generate critical questions
        for scheme in arg_graph.schemes:
            cqs =
self.ontology.get_critical_questions(scheme.type)
            scheme.critical_questions = cqs

        # 2. Search for defeaters
        defeaters = []
        for scheme in arg_graph.schemes:
            for cq in scheme.critical_questions:
                answers = self.kb.query(cq)
                if answers:
                    defeater = self.construct_defeater(cq,
answers)
                    defeaters.append(defeater)

        # 3. Build attack graph
        attack_graph = self.build_attack_graph(
            arg_graph, defeaters
        )

        # 4. Apply argumentation semantics
        labeling =
self.compute_grounded_extension(attack_graph)

        # 5. Check proof standards
        for statement in arg_graph.statements:
            statement.status = self.apply_proof_standard(
                statement, labeling
            )

        return EvaluationResult(
            labeling=labeling,
```

```
            defeaters=defeaters,
            status_map={s.id: s.status for s in
arg_graph.statements}
        )

    def compute_grounded_extension(self, graph):
        # Dung-style acceptability semantics
        # ... (implementation details)
        pass
```

11.4 Argument Generator Module

Responsibilities:

1. Given a target conclusion, find supporting arguments
2. Search knowledge base for applicable schemes
3. Construct argument chains (backwards chaining)
4. Optimize for audience commitments

Algorithm Outline:

```
class ArgumentGenerator:
    def __init__(self, scheme_ontology, kb,
audience_model):
        self.ontology = scheme_ontology
        self.kb = kb
        self.audience = audience_model

    def generate(self, goal: Proposition) ->
List[ArgumentChain]:
        # 1. Identify applicable schemes
        candidate_schemes =
self.ontology.find_schemes_with_conclusion_type(
            goal.type
        )

        # 2. For each scheme, check if premises can be
satisfied
        arguments = []
        for scheme in candidate_schemes:
            premises = scheme.get_premises()
```

```
            # Try to satisfy premises from KB or audience
commitments
            bindings = self.find_bindings(premises, goal)

            if bindings:
                arg = SchemeInstance(scheme, bindings,
goal)
                arguments.append(arg)

        # 3. Recursively generate arguments for missing
premises
        complete_chains = []
        for arg in arguments:
            chain = [arg]
            for premise in arg.premises:
                if not
self.is_acceptable_to_audience(premise):
                    sub_args = self.generate(premise)  #
Recursive
                    chain.extend(sub_args)
            complete_chains.append(ArgumentChain(chain))

        # 4. Rank by likelihood of acceptance
        ranked =
self.rank_by_audience_acceptability(complete_chains)

        return ranked

    def find_bindings(self, premise_templates, conclusion):
        # Query KB for facts that match premise patterns
        # ... (implementation using SPARQL or similar)
        pass
```

11.5 Data Models

Core Entities:

```
// Proposition
interface Proposition {
  id: string;
  text: string;
```

```
  type: PropositionType;  // factual, normative, evaluative
  polarity: 'positive' | 'negative' | 'neutral';
  acceptability: AcceptabilityStatus;
}

enum PropositionType {
  FACTUAL_PAST,
  FACTUAL_PRESENT,
  FACTUAL_FUTURE,
  NORMATIVE_ACTION,
  NORMATIVE_OBLIGATION,
  EVALUATIVE_VALUE,
  SPEECH_ACT
}

enum AcceptabilityStatus {
  IN,         // Accepted
  OUT,        // Rejected
  UNDECIDED  // Neither
}

// Scheme Instance
interface SchemeInstance {
  id: string;
  schemeType: SchemeType;
  premises: Map<string, Proposition>;  // role ->
proposition
  conclusion: Proposition;
  implicitComponents: string[];        // IDs of
reconstructed components
  confidence: number;                  // 0-1, from
recognition
}

// Critical Question
interface CriticalQuestion {
  id: string;
  text: string;
  questionType: CQType;
```

```
  targetComponent: string;              // Which premise/
conclusion
  premiseType: PremiseType;             // For burden of
proof
}

enum CQType {
  PREMISE_ATTACK,
  EXCEPTION,
  USAGE_CONDITION,
  COUNTERARGUMENT_POINTER
}

// Argument Graph
interface ArgumentGraph {
  statements: Proposition[];
  schemes: SchemeInstance[];
  attacks: Attack[];
  supports: Support[];
}

interface Attack {
  from: string;  // attacker ID
  to: string;    // target ID
  type: 'undermine' | 'rebut' | 'undercut';
}

interface Support {
  from: SchemeInstance;
  to: Proposition;
  strength: number;  // Weight/confidence
}
```

12. Implementation Priorities & Roadmap

Phase 1: Foundation (Months 1-3)

Goal: Core scheme library and basic recognition

Deliverables:

1. Scheme ontology (OWL) with top 10 schemes
    - Expert opinion
    - Practical reasoning
    - Cause to effect
    - Consequences (positive/negative)
    - Example
    - Classification
    - Values
    - Commitment
    - Sign
    - Analogy
2. Data models and API
    - Proposition, SchemeInstance, ArgumentGraph classes
    - JSON serialization
    - Basic CRUD operations
3. Rule-based scheme recognizer
    - Keyword/pattern matching
    - Proposition type classification
    - Simple premise-conclusion linking
4. Critical question generator
    - Database of CQs for each scheme
    - Parameterized question templates
5. Argument visualizer
    - Graph rendering (nodes for statements, edges for inference)
    - Scheme type labels
    - Attack/support differentiation

Success Metrics:

- Recognize schemes in 70%+ of simple arguments
- Generate appropriate CQs for identified schemes
- Render argument graphs for manual analysis

Phase 2: Intelligence (Months 4-6)

Goal: ML-based recognition and evaluation

Deliverables:

1. Training pipeline
    - Data collection from AIFdb
    - Feature extraction
    - Model training (Random Forest, Neural Network)

2. Enhanced recognition
   - ML classification of schemes
   - Confidence scoring
   - Ambiguity handling (multiple hypothesis)
3. Enthymeme reconstruction
   - Implicit premise detection
   - Schema-driven completion
   - Semantic similarity for bridging
4. Basic evaluation engine
   - Attack graph construction
   - Grounded semantics computation
   - Acceptability labeling
5. Expanded scheme library (30+ schemes)

Success Metrics:

- Recognition accuracy 80%+ on test set
- Reconstruct 60%+ of implicit premises correctly
- Correctly evaluate argument acceptability in simple cases

Phase 3: Advanced Reasoning (Months 7-9)

Goal: Argument generation and formal frameworks

Deliverables:

1. Argument generator
   - Backwards chaining from goal
   - Knowledge base integration
   - Audience model for premise selection
2. Formal framework integration
   - ASPIC+ implementation or
   - Carneades integration
3. Advanced evaluation
   - Proof standards
   - Burden of proof handling
   - Cumulative arguments
4. Domain-specific modules
   - Legal reasoning templates
   - Scientific argumentation patterns
5. Argument mining pipeline

- End-to-end text → argument graph
- Batch processing

Success Metrics:

- Generate novel arguments achieving goals in 70%+ cases
- Match human evaluation on argument acceptability
- Mine arguments from real documents with F1 > 0.75

Phase 4: Applications (Months 10-12)

Goal: Production-ready systems for specific domains

Deliverables:

1. Debate system
    - Multi-agent argumentation
    - Turn-taking
    - Argument invention
    - Counter-argument generation
2. Educational tool
    - Argument diagramming interface
    - Scheme identification exercises
    - Critical question training
    - Assessment and feedback
3. Legal AI assistant
    - Case analysis
    - Precedent retrieval
    - Statute interpretation
    - Brief generation support
4. Fact-checking tool
    - Claim extraction
    - Evidence assessment
    - Expert opinion evaluation
    - Credibility scoring

Success Metrics:

- User studies showing learning gains
- Legal professionals adopt tool
- Fact-checking accuracy competitive with human raters

13. Testing & Evaluation Strategy

13.1 Unit Testing

Scheme Recognition:

```
def test_expert_opinion_recognition():
    text = "Dr. Smith, a leading virologist, states that
the vaccine is safe."
    schemes = recognizer.recognize(text)

    assert len(schemes) == 1
    assert schemes[0].type == SchemeType.EXPERT_OPINION
    assert "Dr. Smith" in
schemes[0].premises['expert'].text
    assert "vaccine is safe" in schemes[0].conclusion.text

def test_critical_question_generation():
    scheme = create_expert_opinion_instance(...)
    cqs = evaluator.generate_critical_questions(scheme)

    assert len(cqs) == 6
    assert any("credible" in cq.text for cq in cqs)
    assert any("reliable" in cq.text for cq in cqs)
```

Evaluation:

```
def test_argument_defeat_by_premise_attack():
    arg = create_argument(...)
    attack = create_premise_attack(arg.premises[0])

    result = evaluator.evaluate_with_attacks([arg],
[attack])

    assert arg.conclusion.status == AcceptabilityStatus.OUT

def test_undercutting_vs_rebutting():
    arg = create_causal_argument(...)
    undercutter = create_exception_attack(...)

    result = evaluator.evaluate_with_attacks([arg],
[undercutter])

    # Undercutter breaks inference but doesn't deny
conclusion directly
```

```
    assert arg.conclusion.status ==
AcceptabilityStatus.UNDECIDED
```

13.2 Integration Testing

End-to-End Pipeline:

```
def test_mining_to_evaluation():
    text = """
    The government should invest in renewable energy
because
    climate change is causing severe damage, and renewables
    reduce carbon emissions. Dr. Jones, an climate
scientist,
    confirms this link.
    """

    # Mine arguments
    arg_graph = pipeline.analyze(text)

    # Verify structure
    assert len(arg_graph.schemes) >= 2  # Practical
reasoning + expert opinion
    assert any(s.type == SchemeType.PRACTICAL_REASONING for
s in arg_graph.schemes)

    # Evaluate
    result = evaluator.evaluate(arg_graph)

    # Check critical questions generated
    assert len(result.critical_questions) > 0
```

13.3 Human Evaluation

Metrics:

1. **Precision**: Of recognized schemes, % correctly identified
2. **Recall**: Of actual schemes in text, % found
3. **F1 Score**: Harmonic mean of precision and recall
4. **Agreement**: Inter-annotator agreement (Cohen's κ)

Datasets:

- AIFdb corpora (with gold standard annotations)
- Microtext corpus (for structural accuracy)

- Custom domain corpus (for specialized applications)

Benchmarks:

- Feng & Hirst 2011: 0.64-0.98 classification accuracy
- Lawrence & Reed 2015: F1 of 0.59-0.91 (component), 0.62-0.88 (schemes)

Target: Match or exceed these on equivalent tasks.

13.4 Argument Quality Assessment

Beyond recognition, test whether system's judgments align with human evaluation:

Protocol:

1. Present argument to both system and humans
2. System labels: acceptability, strengths, weaknesses
3. Humans judge: accept/reject, strong/weak points
4. Compare:
    - Agreement on acceptability
    - Overlap in critical questions raised
    - Similarity of identified defeaters

Metrics:

- % agreement on accept/reject
- Jaccard similarity on identified issues
- Kendall's τ on argument ranking

14. Critical Implementation Considerations

14.1 Handling Ambiguity

Challenge: Same text may fit multiple schemes.

Example:

```
"You should exercise because it's healthy."
```

Could be:

- Practical reasoning (goal: health → means: exercise)
- Argument from consequences (exercise → positive outcome: health)
- Argument from values (health is valuable → action promoting health is good)

Solutions:

1. **Return Multiple Hypotheses**:

```
interface RecognitionResult {
  hypotheses: Array<{
```

```
    scheme: SchemeInstance;
    confidence: number;
    evidence: string[];
  }>;
}
```

2. **Contextual Disambiguation**:
- Use broader discourse context
- Consider speaker's goals/commitments
- Apply domain constraints
3. **Interactive Refinement**:
- Let user select among alternatives
- Learn from user choices to improve model

14.2 Computational Complexity

Challenge: Argument generation requires search through large space.

Mitigations:

1. **Heuristic Search**:
   - Prioritize schemes based on goal type
   - Use A* with audience acceptability as heuristic
2. **Caching**:
   - Store common argument patterns
   - Reuse sub-arguments
3. **Depth Limits**:
   - Limit chain length in backwards chaining
   - Trade completeness for tractability
4. **Parallel Processing**:
   - Explore multiple scheme hypotheses concurrently
   - Distribute defeater search

14.3 Knowledge Base Requirements

For Argument Generation: Need rich KB of facts, values, commitments.

Sources:

- Structured: DBpedia, Wikidata, ConceptNet
- Unstructured: Wikipedia, news corpus
- Domain-specific: Legal databases, scientific literature

Integration:

```
interface KnowledgeBase {
```

```
  query(pattern: PropositionPattern): Proposition[];
  addFact(prop: Proposition, source: Source): void;
  getCommitments(agent: Agent): Set<Proposition>;
  getValues(agent: Agent): ValueHierarchy;
}
```
Challenge: Keeping KB current and consistent.

Solutions:

- Automated extraction from text (NLP pipeline)
- Incremental updates with provenance tracking
- Conflict resolution strategies

14.4 Explanation Generation

Users need to understand:

- Why was this scheme identified?
- How was this argument evaluated?
- Where did defeaters come from?

Implementation:

```
interface Explanation {
  schemeJustification: {
    scheme: SchemeType;
    reasons: string[];  // "Keyword 'expert' detected",
"Premise matches expert type"
    alternatives: SchemeType[];  // Other considered
schemes
  };

  evaluationTrace: {
    acceptability: AcceptabilityStatus;
    defeaters: Array<{
      defeater: Attack;
      source: 'critical_question' | 'counterargument' |
'inconsistency';
      reason: string;
    }>;
    proofStandard: string;
  };
```

```
  recommendedActions?: string[];  // "Consider addressing CQ3", "Find more expert support"
}
```
Natural Language Generation:

- Template-based: "This argument was identified as Expert Opinion because..."
- Learned: Train NLG model on explanations
- Hybrid: Templates with learned content selection

15. Future Directions & Research Opportunities

15.1 Multi-Modal Argumentation

Beyond Text:

- Images as visual arguments (e.g., graphs showing trends)
- Video analysis (debate, political speeches)
- Multimodal integration (text + image in social media)

Research Questions:

- What schemes apply to visual arguments?
- How do modalities combine in hybrid arguments?

15.2 Cross-Lingual Argumentation

Opportunity: Argument mining in multiple languages.

Approaches:

- Multilingual models (mBERT, XLM-R)
- Parallel corpus construction (like Microtext)
- Universal scheme ontology (language-independent)

Use Case: Analyze international political debates, compare argumentation across cultures.

15.3 Personalization & Adaptation

Goal: Tailor argument generation to specific audiences.

Requirements:

- User modeling (values, beliefs, commitments)
- Adaptive explanation (complexity level)
- Cultural awareness (argumentation norms vary)

Research: How do different groups respond to different schemes?

15.4 Adversarial Robustness

Challenge: Systems can be fooled by adversarial arguments.

Example:

- Surface markers of expert opinion without substance
- Pseudo-causal language without actual mechanism

Solutions:

- Semantic verification (do premises actually support conclusion?)
- External knowledge grounding (fact-checking)
- Adversarial training

15.5 Real-Time Dialogue Systems

Application: Live debate assistants, negotiation support.

Requirements:

- Low latency scheme recognition
- Dynamic argument graph updates
- Proactive counter-argument suggestion
- Turn-taking and dialogue move classification

Research: Integration with dialogue management, speech recognition, real-time NLU.

16. Key Takeaways for Developers

Must-Dos:

1. **Start with Purpose-Based Classification** (Section 3.1)
    - Clearest organizing principle
    - Maps directly to use cases
    - Guides UI design
2. **Implement Tier 1 Schemes First** (Section 4.1)
    - Cover 61% of natural arguments
    - Well-studied with clear structure
    - Highest ROI
3. **Model Critical Questions Properly** (Section 6)
    - Critical for evaluation
    - Different types require different handling
    - Burden of proof allocation is key
4. **Think in Networks, Not Isolated Arguments** (Section 5)
    - Real arguments are composite

    - Schemes interconnect
    - Enthymeme reconstruction essential
5. **Use Formal Frameworks** (Section 7)
    - Don't reinvent evaluation semantics
    - Carneades or ASPIC+ provide solid foundation
    - Open source resources available
6. **Leverage Ontologies** (Section 10)
    - Formal specification enables automation
    - Inheritance and reasoning for free
    - Use existing: http://arg.tech/aif.owl
7. **Learn from Argument Mining Research** (Section 9)
    - Keywords/patterns + ML hybrid works
    - Scheme structure constrains proposition types
    - Use AIFdb for training data

Common Pitfalls to Avoid:

1. **Confusing Schemes with Logical Forms**
    - Schemes merge content + inference
    - Same semantic relation, multiple logical forms
2. **Treating All Premises Equally**
    - Ordinary, assumptions, exceptions have different burden of proof
    - Critical for proper evaluation
3. **Ignoring Implicitness**
    - Real arguments leave much unstated
    - Must reconstruct enthymemes
4. **Over-Simplifying Evaluation**
    - It's not just validity checking
    - Defeasibility, attack types, proof standards all matter
5. **Neglecting Domain Specificity**
    - Legal, scientific, political argumentation have distinct patterns
    - Generic schemes need domain adaptation

17. Additional Resources

Code Repositories:

- **Carneades**: https://github.com/carneades
- **AIFdb**: http://aifdb.org

- **OVA**: https://ova.arg.tech

Tools:

- **Carneades 4**: http://carneades.fokus.fraunhofer.de/carneades
- **OVA+**: http://ova.arg.tech
- **Argument Web Services**: http://www.argumenta tion.org

Data:

- **AIFdb Corpora**: http://corpora.aifdb.org
- **Potsdam Microtext**: https://github.com/peldszus/arg-microtext

Ontologies:

- **AIF Ontology**: http://arg.tech/aif.owl
- **Scheme Ontology**: Embedded in AIF

Key Papers:

- Walton, Reed, Macagno (2008): *Argumentation Schemes* [foundational]
- Feng & Hirst (2011): Scheme classification [ML approach]
- Lawrence & Reed (2015): Scheme-guided mining [state-of-the-art]
- Gordon & Walton (2016): Carneades model [formal framework]
- Prakken et al. (2015): ASPIC+ with schemes [legal reasoning]

Appendix A: Complete Scheme Reference (Top 30)

For each scheme, provide:

- Full structure (premises, conclusion)
- All critical questions
- Example instantiations
- Implementation notes

Format:

```
SCHEME NAME
Type: [Inductive/Deductive/Abductive/Practical]
Classification: [Purpose-based category]

Structure:
Premise 1: ...
Premise 2: ...
...
Conclusion: ...

Critical Questions:
```

```
CQ1: ...
CQ2: ...
...

Example:
[Natural language example]
[Formalized instance]

Implementation Notes:
- Keywords: ...
- Proposition types: ...
- Common variations: ...
- Attack patterns: ...

Related Schemes:
- Generalizes: ...
- Specializes: ...
- Combines with: ...
```
(Detailed reference for 30 schemes would follow...)

Appendix B: Keyword/Pattern Database

Organized by scheme, comprehensive lists of:

- Trigger words
- Syntactic patterns
- Discourse markers
- Semantic roles

For ML feature engineering and rule-based components.

Appendix C: Test Suite

Collection of:

- Unit test cases for each scheme
- Integration test scenarios
- Gold standard annotated examples
- Edge cases and ambiguities

For validation and regression testing.

Document Metadata

Source: Macagno, F., Walton, D., Reed, C. (2017). "Argumentation Schemes: History, Classifications, and Computational Applications." *IFCoLog Journal of Logics and Their Applications*, 4(8).

Analysis Date: 2025

Intended Audience: Software developers, ML engineers, computational linguists, AI researchers implementing argumentation systems

Maintenance: Living document; update as systems evolve and new research emerges

Version: 1.0

Contact: [Your implementation team contact]

This comprehensive guide synthesizes the theoretical foundations and practical applications of argumentation schemes for direct implementation. The modular structure allows teams to:

1. **Understand** the conceptual basis (Sections 1-3)
2. **Prioritize** implementation (Section 4, 12)
3. **Architect** systems (Section 11)
4. **Test** rigorously (Section 13)
5. **Extend** to applications (Section 8)

The paper's key contribution—showing how schemes evolved, how they should be classified, and how they interconnect in networks—provides the roadmap for building sophisticated argumentation AI.