Summary of "Flat Syntax" Paper: Computational Implementation Focus

Core Proposal

William Croft proposes a **non-recursive, three-level syntactic annotation scheme** designed for practical descriptive linguistics that could be computationally implemented. The scheme uses boundary markers (similar to morphological annotation) rather than complex tree structures.

Key Components for Implementation

1. Boundary Markers (§2.2, §2.6)

The annotation uses ASCII-compatible symbols on standard keyboards:

- **Space** - separates phrasal CEs (words/phrase elements)
- + - separates clausal CEs (phrases within clauses)
- # - separates sentential CEs (clauses within sentences)
- `.` - sentence boundary
- `{ }` - marks interrupting/center-embedded units (analogous to infixes)
- ^ - joins fixed contiguous multiword expressions

Example annotation:

```
The two brothers + might + lose + the game # and + exit + the competition .
```

2. Three Fixed Syntactic Levels (§2.3)

Critical simplification: No recursion means exactly three levels:

- **Sentence level** - consists of clauses, phrases, or words
- **Clause level** - consists of phrases or words
- **Phrase level** - consists of words

This creates a **flat, non-hierarchical structure** amenable to tabular data formats.

3. Data Format: CLDF-Compatible (§1.2, §2.6)

The scheme is designed to work with:

• **CSV/TSV tabular format** (Cross-Linguistic Data Format)

• Plain text with comma/tab-separated values

• Compatible with existing morphological glossing (Leipzig Glossing Rules)

• Boundary markers append to words (e.g., `competition .` or `game #`)

CLDF Structure:

```
Analyzed Text: word1 + word2 + word3 # word4 + word5 .
Gloss: GLOSS1 GLOSS2 GLOSS3 GLOSS4 GLOSS5
Translation: Free translation here
```

4. Construction Element (CE) Labels (§3)

Optional three-tier annotation providing syntactic roles:

Phrasal CE Labels:

• **Head** - head of phrase

• **Mod** - modifier (articles, adjectives, demonstratives, etc.)

• **Adm** - admodifier (modifies modifiers, e.g., "very" in "very large")

• **Strategy labels**: Adp (adposition), Lnk (linker), Clf (classifier), Idx (index), Conj (conjunction)

Clausal CE Labels:

• **Pred** - predicate

• **Arg** - argument phrase (subject, object, oblique - undifferentiated)

• **CPP** - complex predicate part

• **Gen** - genitive phrase modifier

• **FPM** - flagged phrase modifier (spatial/temporal)

• **Conj** - conjunction

Sentential CE Labels:

• **Main** - main clause

• **Adv** - adverbial clause

• **Rel** - relative clause

• **Comp** - complement clause

• **Dtch** - detached phrase (topic/focus)

• **Int** - interactive (discourse markers, interjections)

Table Format Example (§3.1, Table 1):

```
Analyzed Text:  The two brothers + might + lose # and + exit + the competition .
Phrasal CEs:    Mod Mod Head      Head    Head   Conj  Head   Mod  Head          Head
Clausal CEs:    Arg               CPP     Pred   Conj  Pred   Arg
Sentential CEs: Main                              Main
```

5. Automatic CE Annotation Strategy (§3.2)

The paper proposes **automatic generation** of CE labels from:

Input Sources:

1. **Flat syntax boundary markers** (from §2)
2. **Interlinear morpheme translation** (IMT/gloss line)
3. **Lexicon with word class information**
4. **Basic word order patterns** (SOV/SVO, GenN/NGen, etc.)

Computational Approach:

Phrasal Level Detection:

• Closed-class function words (determiners, classifiers) → strategy labels

• Morphological features (case, number, gender) → identify Heads

• Word order + word class → distinguish Mod from Head

• Adjacency patterns → identify Adm (e.g., Adm-Adj sequence)

Clausal Level Detection:

• TAM (tense-aspect-modality) morphology → identify Pred

• Case marking/adpositions + word order → identify Arg vs. Gen vs. FPM

• Auxiliary class words → identify CPP

• Conjunctions → identify coordination

Sentential Level Detection:

• Subordinating conjunctions/morphology → Adv, Comp, Rel

• Relativizers, complementizers → specific clause types

• Deranked verb forms (participles, gerunds) → subordinate clauses

• Default (no subordination marking) → Main

Table 6-8 (§3.2) provide systematic mappings:

• Grammatical concepts → function word classes → bound morphemes

• These create **feature-based rules** for automatic annotation

6. Handling Complex Cases

Interruption (§2.4):

```
The tree {that + fell + on {my} house} had + died + last winter .
```

• Opening `{` attaches to first word of interrupting unit

• Closing `}` attaches to last word of interrupting unit

• Preserves word order while marking discontinuity

Multiword Expressions (§2.5):

```
on^top^of, by^and^large
```

• Treated as single lexical units

• No spaces within MWE

• Single gloss in IMT line

7. Computational Advantages

1. **Regular structure**: Fixed three levels enable array/matrix representation
2. **No recursive parsing**: Simpler computational complexity
3. **Text-based**: Easy storage, version control, human-readable
4. **Incremental annotation**: Can add CE labels after basic boundaries
5. **Machine learning ready**: Tabular format suitable for supervised learning
6. **Cross-linguistic**: Not language-specific like some treebanks

8. Conversion Potential (§4)

The paper notes (§4, end) that:

• Conversion to **Universal Dependencies** is "fairly straightforward" for headed constructions

• CE role labels map to UD relations (with some generalization)

• Head-dependent structure can be recovered from CE annotations + word order

Implementation Workflow

```
1. Input: Glossed text with morphological segmentation
   ↓
```

```
2. Add boundary markers (+, #, {}, ^)
   ↓
3. Store in CLDF tabular format
   ↓
4. (Optional) Extract features from:
   - Glosses (morphological categories)
   - Lexicon (word classes)
   - Grammar (word order patterns)
   ↓
5. (Optional) Auto-generate CE labels
   ↓
6. Output: Three-tier annotated text
```

Data Requirements for Automation

To implement automatic CE labeling:

• **Standardized gloss abbreviations** (Leipzig Glossing Rules)

• **Lexicon with word classes** (N, V, Adj, Adv, Adp, etc.)

• **Word order typology** (SOV/SVO, GenN/NGen, RelN/NRel, etc.)

• **Closed-class word inventories** (conjunctions, auxiliaries, etc.)

• **Morphological feature specifications** (case, TAM, indexation)

Key Technical Insight (§4)

The theoretical discussion in §4 provides computational justification:

• **Empirical constraint**: Center-embedding depth rarely exceeds 2-3 levels (Karlsson 2007-2010)

• **Text frequency**: Deep recursion extremely rare in spoken language

• **Processing motivation**: Flat structure reflects actual language use patterns

• **Alternative strategies**: Languages evolve mechanisms to avoid deep embedding

This suggests the flat model isn't just practically simpler—it may be **cognitively adequate**.

Flat Syntax Annotation Pipeline: Practical Implementation Strategy

Overview of Proposed Architecture

```
Input Text
    ↓
[1] UD Parser (spaCy/Stanza/etc.)
    ↓
[2] UD-to-Flat Syntax Converter
    ↓
[3] LLM-based Refinement (few-shot)
```

```
    ↓
Output: Flat Syntax Annotation
```

PHASE 1: Universal Dependencies as Foundation

What UD Provides (Directly Usable)

UD Feature	Flat Syntax Utility	Mapping Strategy
POS tags	Word class for CE label assignment	NOUN/PROPN→Head, ADJ→Mod, ADP→Adp, etc.
Morphological features	Phrasal CE detection	Case, Number, Gender→Head identification
Dependency relations	Clausal CE roles	nsubj/obj/obl→Arg, root→Pred, acl→Rel, advcl→Adv
Head-dependent structure	Phrase boundaries	Subtree spans = phrases
Token order	Sequence preservation	Direct mapping to flat annotation

Critical UD Information for Flat Syntax

```
# Example UD output structure (CoNLL-U format)
{
    "id": 3,
    "form": "tree",
    "lemma": "tree",
    "upos": "NOUN",           # ← Word class
    "xpos": "NN",
    "feats": {                # ← Morphology
        "Number": "Sing"
    },
    "head": 6,                # ← Parent in dependency tree
    "deprel": "nsubj",        # ← Syntactic relation
    "deps": None,
    "misc": None
}
```

Key advantages:

• UD parsers are mature, multilingual, reasonably accurate

• Morphological features explicitly marked

• Syntactic relations provide **strong cues** for CE labels

• Open-source models available (spaCy, Stanza, Trankit)

PHASE 2: UD-to-Flat Syntax Conversion Rules

2.1 Phrase Boundary Detection

Algorithm:

```
For each token in UD parse:
    1. Get dependency subtree (token + all descendants)
    2. If subtree forms a valid phrase constituent:
       - Mark with '+' after last token in subtree
    3. Special cases:
       - Compound/multiword expressions: use '^'
       - Coordinated phrases: insert '+' between coordinands
```

UD Relations → Phrase Boundaries:

UD Pattern	Flat Syntax Boundary	Example
Token with no children	Single-word phrase	`dog +`
Token + {det, amod, nummod, ...}	Multi-word phrase	`the big dog +`
Token + case (preposition)	Prepositional phrase	`in the house +`
conj relation	Separate coordinated phrases	`cats + and dogs +`

2.2 Clause Boundary Detection

Algorithm:

```
For each predicate (VERB/AUX with deprel=root or linked to root):
    1. Collect all direct arguments (nsubj, obj, obl, xcomp, ccomp)
    2. Mark clause boundary '#' after last element
    3. Handle subordination:
       - acl/acl:relcl → Rel clause
       - advcl → Adv clause
       - ccomp/xcomp → Comp clause
```

UD Relations → Clause Boundaries:

UD Relation	Flat Syntax Pattern	Sentential CE Label
`root`	Main clause	Main
`acl:relcl`	Separate clause with #	Rel
`advcl`	Separate clause with #	Adv
`ccomp`	Separate clause with #	Comp
`parataxis`	Separate clause with #	Main (coordination)

2.3 CE Label Mapping (Rule-Based)

Phrasal CE Labels from UD

```
PHRASAL_CE_MAPPING = {
    # Direct mappings from UPOS
```

```
    "DET": "Mod",
    "ADJ": "Mod",
    "NUM": "Mod",
    "ADP": "Adp",
    "CCONJ": "Conj",  # if coordinating within phrase

    # Context-dependent (needs deprel)
    ("ADV", "advmod"): "Adm",  # if modifying adjective/adverb
    ("NOUN", "nmod:poss"): "Gen",  # possessive
    ("PRON", "nmod:poss"): "Gen",

    # Default for content words
    "NOUN": "Head",
    "PROPN": "Head",
    "PRON": "Head",
    "VERB": "Head",  # when nominalized
}
```

Clausal CE Labels from UD

```
CLAUSAL_CE_MAPPING = {
    # Predicate
    ("VERB", "root"): "Pred",
    ("VERB", "ccomp"): "Pred",  # in complement clause
    ("VERB", "acl"): "Pred",     # in relative clause

    # Arguments (phrases)
    "nsubj": "Arg",
    "obj": "Arg",
    "iobj": "Arg",
    "obl": "Arg",

    # Complex predicate parts
    "aux": "CPP",
    "cop": "CPP",
    ("ADV", "advmod"): "CPP",  # if modifying verb directly

    # Modifiers (phrases)
    "nmod:poss": "Gen",
    ("nmod", "+case"): "FPM",  # flagged phrase modifier

    # Coordination
    "cc": "Conj",
}
```

Sentential CE Labels from UD

```
SENTENTIAL_CE_MAPPING = {
    "root": "Main",
    "acl:relcl": "Rel",
    "acl": "Rel",  # often relative-like
    "advcl": "Adv",
    "ccomp": "Comp",
    "xcomp": "Comp",
    "parataxis": "Main",  # coordinate main clause

    # Special cases
    "vocative": "Int",
    "discourse": "Int",
    "dislocated": "Dtch",
}
```

2.4 Handling Center-Embedding (Interruption)

Detection Algorithm:

```
def detect_interruption(tree):
    """
    Detect if a clause/phrase interrupts its parent.

    UD clue: Child clause tokens appear between
             parent clause's tokens (non-contiguous span)
    """
    for node in tree:
        parent_span = get_span(node.head)
        child_span = get_span(node)

        if child_span.start > parent_span.start and \
           child_span.end < parent_span.end:
            # Child interrupts parent
            mark_with_braces(node)  # { ... }
```

Example:

```
UD: The tree [that fell on my house]_acl:relcl had died
          ↓
Flat: The tree + {that + fell + on {my} house} had + died .
```

UD provides:

• `acl:relcl` attached to "tree" (head=2)

• Relative clause tokens (4-8) between "tree" (2) and "had" (9)

• **Non-contiguous span** = interruption

2.5 Multiword Expressions

UD provides:

• `fixed` relation for fixed MWEs

• `flat` for names

• `compound` for some compounds

Conversion:

```
if token.deprel in ["fixed", "flat", "compound"]:
    join_with_caret(token, token.head)  # use '^'
```

Example:

```
UD: on → top ← of (fixed)
       ↓
Flat: on^top^of
```

PHASE 3: LLM-Based Refinement Pipeline

Why LLM is Needed (UD Limitations)

Problem	UD Limitation	LLM Solution
Ambiguous deprels	UD `obl` could be Arg or FPM	Context-aware disambiguation with examples
Complex predicates	UD doesn't mark CPP consistently	Semantic understanding of auxiliary vs. main verb
Pragmatic labels	Main vs. Adv based on assertion (not morphosyntax)	Pragmatic reasoning
Cross-linguistic variation	UD annotations vary by treebank	Few-shot learning for consistency
Interrupted constructions	Complex nested dependencies	Spatial reasoning about spans

3.1 Few-Shot Prompting Strategy

Template Structure:

```
You are a linguistic annotator specializing in Flat Syntax annotation.

# Annotation Principles
[Insert Croft's key principles from paper]

# Input Format
You receive:
1. Original sentence
2. UD parse (CoNLL-U format)
3. Preliminary flat syntax annotation

# Your Task
Review and correct the preliminary annotation following these rules:
[Specific rules for the ambiguous case]

# Examples (Few-Shot)
[3-5 gold-standard examples of the ambiguous pattern]

# Current Sentence
[Sentence to annotate]

# Preliminary Annotation
[Rule-based output from Phase 2]

# Output
Corrected annotation with explanation of changes.
```

3.2 Key Disambiguation Tasks for LLM

Task 3.2.1: Arg vs. FPM (Oblique Arguments vs. Modifiers)

Problem:

• UD labels both as `obl`

• Flat Syntax distinguishes: Arg (participant) vs. FPM (modifier)

Few-Shot Prompt:

```
Distinguish participant arguments (Arg) from flagged phrase modifiers (FPM):

Example 1:
Sentence: "She walked to school."
UD: walked → to school [obl]
Flat: She + walked + to school .
       Arg   Pred      Arg
```

```
Reason: "to school" is the destination (participant/goal of walking)

Example 2:
Sentence: "The book on the table is red."
UD: book → on the table [nmod]
Flat: The book + on the table + is + red .
       Mod Head   Adp Mod  Head   CPP  Head
Clausal: Arg       FPM             Pred
Reason: "on the table" modifies "book" (location modifier, not participant)

Example 3:
Sentence: "He put the book on the table."
UD: put → on the table [obl]
Flat: He + put + the book + on the table .
      Arg  Pred  Arg        Arg
Reason: "on the table" is a required location argument of "put"

Now analyze: [New sentence]
```

LLM advantages:

• Semantic role understanding (participant vs. circumstance)

• Valency pattern knowledge

• Context sensitivity

Task 3.2.2: Main vs. Adv (Pragmatic Assertion)

Problem:

• Morphosyntactic subordination ≠ pragmatic non-assertion

• Some subordinate forms express main events (insubordination)

• Some coordinate structures have asymmetric assertion

Few-Shot Prompt:

```
Identify pragmatically asserted clauses (Main) vs. backgrounded clauses (Adv):

Example 1:
Sentence: "After I ate, I left."
Annotation: after + I + ate # I + left .
Sentential: Adv            Main
Reason: "I ate" provides temporal background; "I left" is the main assertion

Example 2:
Sentence: "I ate and left."
Annotation: I + ate # and + left .
Sentential: Main      Main
Reason: Both events equally asserted (coordinate)

Example 3 (Insubordination):
Sentence: "If you could just sign here..." [request context]
Annotation: If you + could + just + sign + here .
Sentential: Main
Reason: Despite "if" (subordinator), this is a main speech act (polite request)

Example 4:
Sentence: "He worked at the mountain and she tended the store." [Japanese deranking]
Annotation: He + worked + at the mountain # and + she + tended + the store .
Sentential: Main                          Main
Reason: Both events asserted despite deranking morphology (see Croft §4.1)

Now analyze: [New sentence]
```

LLM advantages:

• Pragmatic reasoning (assertion vs. presupposition)

• Context understanding (speech acts)

• Semantic relation detection (cause, time, condition)

Task 3.2.3: Pred vs. CPP (Complex Predicate Analysis)

Problem:

• What counts as "primary predicate" vs. "complex predicate part"?

• Auxiliaries, modals, aspectuals vary cross-linguistically

Few-Shot Prompt:

```
Identify primary predicate (Pred) vs. complex predicate parts (CPP):

Example 1:
Sentence: "She might have been running."
Annotation: She + might + have + been + running .
Clausal: Arg   CPP     CPP    CPP    Pred
Reason: "running" is semantic core; "might/have/been" are TAM auxiliaries

Example 2:
Sentence: "He also wrecked your car."
Annotation: He + also + wrecked + your car .
Clausal: Arg  CPP    Pred      Arg
Reason: "wrecked" is semantic core; "also" modifies the predication

Example 3:
Sentence: "The tree had died."
Annotation: The tree + had + died .
Clausal: Arg       CPP   Pred
Reason: "died" is semantic core; "had" marks past perfect aspect

Principle: The primary predicate expresses the main event/state.
CPPs express TAM, manner, or other modifications of the predication.

Now analyze: [New sentence]
```

Task 3.2.4: Interruption Detection (Center-Embedding)

Problem:

• Complex nested structures may not be caught by span analysis

• Need to recognize when braces { } are required

Few-Shot Prompt:

```
Mark interrupted constructions with braces { }:

Example 1:
Sentence: "The tree that fell on my house had died."
Without braces: The tree + that + fell + on my house + had + died .
With braces: The tree + {that + fell + on {my} house} had + died .
Reason: Relative clause interrupts main clause; possessive interrupts PP

Example 2:
Sentence: "The man who's picking pears comes down."
Annotation: The man + {who's + picking + pears} comes + down .
Reason: Relative clause interrupts subject phrase of main clause

Example 3 (No interruption):
Sentence: "I saw the man who's picking pears."
```

```
Annotation: I + saw + the man + who's + picking + pears .
Reason: Relative clause follows (doesn't interrupt) main clause

Rule: Use { } when a clause/phrase is surrounded by parts of another
clause/phrase of the same level.

Now analyze: [New sentence]
```

3.3 Implementation: Two-Stage LLM Pipeline

Stage 1: Batch Disambiguation

```
def llm_refinement_stage1(ud_parse, preliminary_flat):
    """
    Focused disambiguation of known problem patterns.
    Fast, targeted prompts.
    """

    problems = detect_ambiguous_patterns(preliminary_flat, ud_parse)
    # Returns: [(token_id, issue_type, context), ...]

    corrections = []
    for problem in problems:
        if problem.type == "ARG_vs_FPM":
            prompt = generate_arg_fpm_prompt(problem, few_shot_examples)
            correction = llm_call(prompt, model="gpt-4", temperature=0)
            corrections.append(correction)

        elif problem.type == "MAIN_vs_ADV":
            prompt = generate_main_adv_prompt(problem, few_shot_examples)
            correction = llm_call(prompt, model="gpt-4", temperature=0)
            corrections.append(correction)

        # ... other disambiguation tasks

    return apply_corrections(preliminary_flat, corrections)
```

Stage 2: Holistic Review

```
def llm_refinement_stage2(stage1_output, ud_parse, original_sentence):
    """
    Full sentence review for consistency and missed patterns.
    More expensive, but catches complex interactions.
    """

    prompt = f"""
    Review this Flat Syntax annotation for correctness:

    Original: {original_sentence}

    Annotation:
    {format_flat_syntax_table(stage1_output)}

    UD Parse (reference):
    {format_ud_parse(ud_parse)}

    Check for:
    1. Phrase boundaries ('+') correctly placed?
    2. Clause boundaries ('#') correctly placed?
    3. Interruptions ('{{ }}') properly marked?
    4. CE labels consistent with Flat Syntax principles?
    5. Any missed multiword expressions ('^')?

    Few-shot examples:
    {load_gold_examples(n=5)}
```

```
    Provide corrected annotation if needed, or confirm if correct.
    """

    return llm_call(prompt, model="gpt-4", temperature=0)
```

PHASE 4: Few-Shot Gold Standard Creation

4.1 Minimal Annotation Set Strategy

Goal: Create a small but maximally informative set of annotated examples.

Selection Criteria:

1. **Coverage of ambiguous patterns** (not random sentences)
2. **Cross-linguistic diversity** (if multilingual)
3. **Complexity gradient** (simple → complex)

Recommended Size:

• **50-100 sentences** for targeted disambiguation (Stage 1)

• **20-30 sentences** for holistic review (Stage 2)

4.2 Annotation Protocol

```
For each gold-standard sentence:

1. Parse with UD parser
2. Apply rule-based conversion (Phase 2)
3. Manual correction by expert linguist
4. Document ambiguous decisions with rationale
5. Store as few-shot example with:
   - Original sentence
   - UD parse
   - Rule-based output
   - Corrected flat syntax
   - Explanation of corrections
```

4.3 Example Categories to Cover

Pattern Category	Example Sentence	Key Ambiguity
Arg vs. FPM	"She walked to school" vs. "The book on the table"	Participant vs. modifier obliques
Main vs. Adv (subordination)	"After I ate, I left"	Temporal backgrounding
Main vs. Adv (coordination)	"I ate and left"	Both asserted
Pred vs. CPP	"She might have been running"	Which verb is primary?
Center-embedding (Rel)	"The tree that fell had died"	Interrupting relative clause

Center-embedding (Gen)	"on my house"	Interrupting possessive
Multiword expression	"on top of", "by and large"	Fixed MWE detection
Complex predicate	"take advantage of"	Lexicalized multi-word predicate
Insubordination	"If you could sign here..."	Subordinate form, main function
Detached topic	"John, I saw him"	Pragmatic detachment
Interactive	"Well, I think so"	Discourse marker

4.4 Structured Few-Shot Examples

Format:

```
{
  "id": "example_001",
  "sentence": "She walked to school.",
  "ud_parse": { ... },
  "rule_based_output": {
    "boundaries": "She + walked + to school .",
    "phrasal_ce": ["Head", "Head", "Adp Mod Head", ""],
    "clausal_ce": ["Arg", "Pred", "Arg?FPM", ""],
    "sentential_ce": ["Main"]
  },
  "gold_annotation": {
    "boundaries": "She + walked + to school .",
    "phrasal_ce": ["Head", "Head", "Adp Mod Head", ""],
    "clausal_ce": ["Arg", "Pred", "Arg", ""],
    "sentential_ce": ["Main"]
  },
  "correction": {
    "type": "ARG_vs_FPM",
    "changed": "to school: FPM → Arg",
    "rationale": "'to school' is the goal/destination argument of 'walk', a required
  participant, not a circumstantial modifier"
  },
  "linguistic_note": "Direction/goal obliques with motion verbs are typically Arg in
  Flat Syntax (participants in the event)"
}
```

PHASE 5: Active Learning Loop (Optional Enhancement)

```
1. Annotate large corpus with rule-based + LLM pipeline
2. Identify low-confidence predictions (LLM temperature variation)
3. Human expert reviews uncertain cases
4. Add confirmed annotations to few-shot pool
5. Retrain/update prompt engineering
6. Iterate
```

Confidence Metrics:

• LLM gives different answers with temperature > 0

• Rule-based heuristics conflict with LLM output

• Multiple equally-scored alternatives in beam search

PHASE 6: Evaluation Metrics

Annotation Quality Metrics

Metric	Definition	Evaluation Level
Boundary Accuracy	% correct '+', '#', '.', '{}' placement	Token-level
CE Label Accuracy	% correct phrasal/clausal/sentential labels	Construction-level
Inter-annotator Agreement	Cohen's κ between human & system	Full annotation
Conversion Fidelity	Agreement with manually annotated gold standard	Sentence-level

Specific Evaluations

1. **Phrase boundary F1**: Precision/recall of '+' placement
2. **Clause boundary F1**: Precision/recall of '#' placement
3. **Interruption detection**: Precision/recall of '{}' usage
4. **Label confusion matrix**: Which CE labels are most confused?

IMPLEMENTATION ROADMAP

MVP (Minimum Viable Product)

Timeline: 2-3 weeks

```
Week 1: UD Parser Integration
- Choose UD parser (spaCy/Stanza)
- Implement CoNLL-U reader
- Build UD → Flat Syntax rule-based converter
- Focus on boundary markers only (no CE labels yet)

Week 2: Rule-Based CE Labeling
- Implement phrasal/clausal/sentential label mappings
- Handle basic cases (80% coverage expected)
- Create test suite (20-30 sentences)

Week 3: LLM Integration
- Design prompts for top 3 ambiguities
- Create 20 few-shot gold examples
- Implement Stage 1 refinement
- Evaluate on test set
```

Full System

Timeline: 2-3 months

```
Month 1: MVP + Gold Standard Creation
- Expand gold standard to 100 examples
- Cover all ambiguity patterns
- Add cross-linguistic examples (if multilingual)

Month 2: Advanced LLM Pipeline
- Implement Stage 2 holistic review
- Add active learning loop
- Optimize few-shot selection algorithm
- Create annotation guidelines document

Month 3: Evaluation & Iteration
- Large-scale evaluation on test corpus
- Compare with manual annotations
- Error analysis and rule refinement
- Documentation and API development
```

TECHNICAL STACK RECOMMENDATIONS

Core Components

```
# 1. UD Parsing
import stanza  # or spacy with spacy-udpipe
nlp = stanza.Pipeline('en', processors='tokenize,pos,lemma,depparse')

# 2. Rule-based conversion
from flat_syntax import UDConverter
converter = UDConverter(language='en')
flat_annotation = converter.convert(ud_parse)

# 3. LLM refinement
from openai import OpenAI
client = OpenAI()
refined = refine_with_llm(flat_annotation, few_shot_examples)

# 4. Output formatting
from flat_syntax import CLDFFormatter
cldf_output = CLDFFormatter.to_csv(refined)
```

Alternative LLM Options

Model	Pros	Cons	Use Case
GPT-4	Best accuracy, instruction-following	Expensive, API-dependent	Production pipeline
Claude 3.5	Strong reasoning, long context	API-dependent	Complex cases
Llama 3.1 (70B)	Open-source, free hosting possible	Requires fine-tuning	Cost-conscious deployment
Mixtral 8x7B	Good quality, fast	Less powerful than GPT-4	Real-time annotation

Output Formats

```
# 1. CLDF-compatible CSV
output.to_csv('annotated_corpus.csv', format='cldf')

# 2. JSON for further processing
output.to_json('annotated_corpus.json')

# 3. Human-readable table
output.to_table('annotated_corpus.txt', format='markdown')

# 4. Visualization (HTML)
output.to_html('annotated_corpus.html', interactive=True)
```

COST ESTIMATION (for 10,000 sentences)

Using GPT-4

```
Assumptions:
- 20 tokens/sentence average
- Stage 1: 30% sentences need disambiguation (3,000 sentences)
- Stage 2: 10% need full review (1,000 sentences)

Stage 1 (targeted):
- Prompt: ~500 tokens (few-shot examples)
- Output: ~100 tokens
- 3,000 × 600 tokens = 1.8M tokens
- Cost: ~$18-36 (depending on GPT-4 pricing)

Stage 2 (holistic):
- Prompt: ~1,000 tokens (full sentence + examples)
- Output: ~200 tokens
- 1,000 × 1,200 tokens = 1.2M tokens
- Cost: ~$12-24

Total: ~$30-60 for 10,000 sentences
```

Using Open-Source LLMs (Self-Hosted)

```
One-time setup cost: GPU rental (~$1-2/hour) or local deployment
Per-sentence cost: Essentially free (compute only)
Better for large-scale annotation
```