CognitiveTwin V3 — COG-RLM Scoring & Training Pipeline

Complete Technical Documentation

Comp-Core / RAG++ / ML Layer

Table of Contents

CognitiveTwin V3: Project Overview

> **Version**: 3.0.0
> **Status**: Implementation Phase
> **Last Updated**: 2025-12-31

---

1. Project Goals

1.1. Primary Objective: Eliminate Permission-Seeking Behavior

The fundamental goal of CognitiveTwin V3 is to train a model that executes on directive prompts without asking for unnecessary confirmation.

1.1.1. Current Problem (V2 Behavior)

- Model frequently ends responses with "Would you like me to...?"
- Model asks "Should I...?" when the user's intent is clear
- Model offers options instead of executing ("I can do A, B, or C")
- Model stalls with "Before I proceed..." preambles

1.1.2. Target Behavior (V3)

- Execute immediately when directive is complete
- State assumptions as declarations, not questions
- Produce artifacts when requested without confirmation
- Only ask questions when genuinely blocked

1.2. Secondary Objective: Train Model to Execute on Directive Prompts

1.2.1. Directive Completeness Detection

- Compute a `directive_completeness` score (0.0 - 1.0)
- When score >= 0.7, model must not ask permission

- Score components:
  - +0.35: Imperative verb present ("rewrite", "implement", "generate")
  - +0.25: Output format specified ("in JSON", "as CSV", "don't omit")
  - +0.20: All required inputs present
  - -0.40: Required input missing
  - -0.20: Material ambiguity present

1.2.2. Question Policy Enforcement

- `no_questions`: Execute without asking (directive complete)
- `questions_if_required`: Ask only if blocked on correctness
- `questions_allowed`: Open-ended brainstorming permitted

1.3. Tertiary Objective: Preserve Justified Clarifications Only

1.3.1. Justified Clarification Criteria

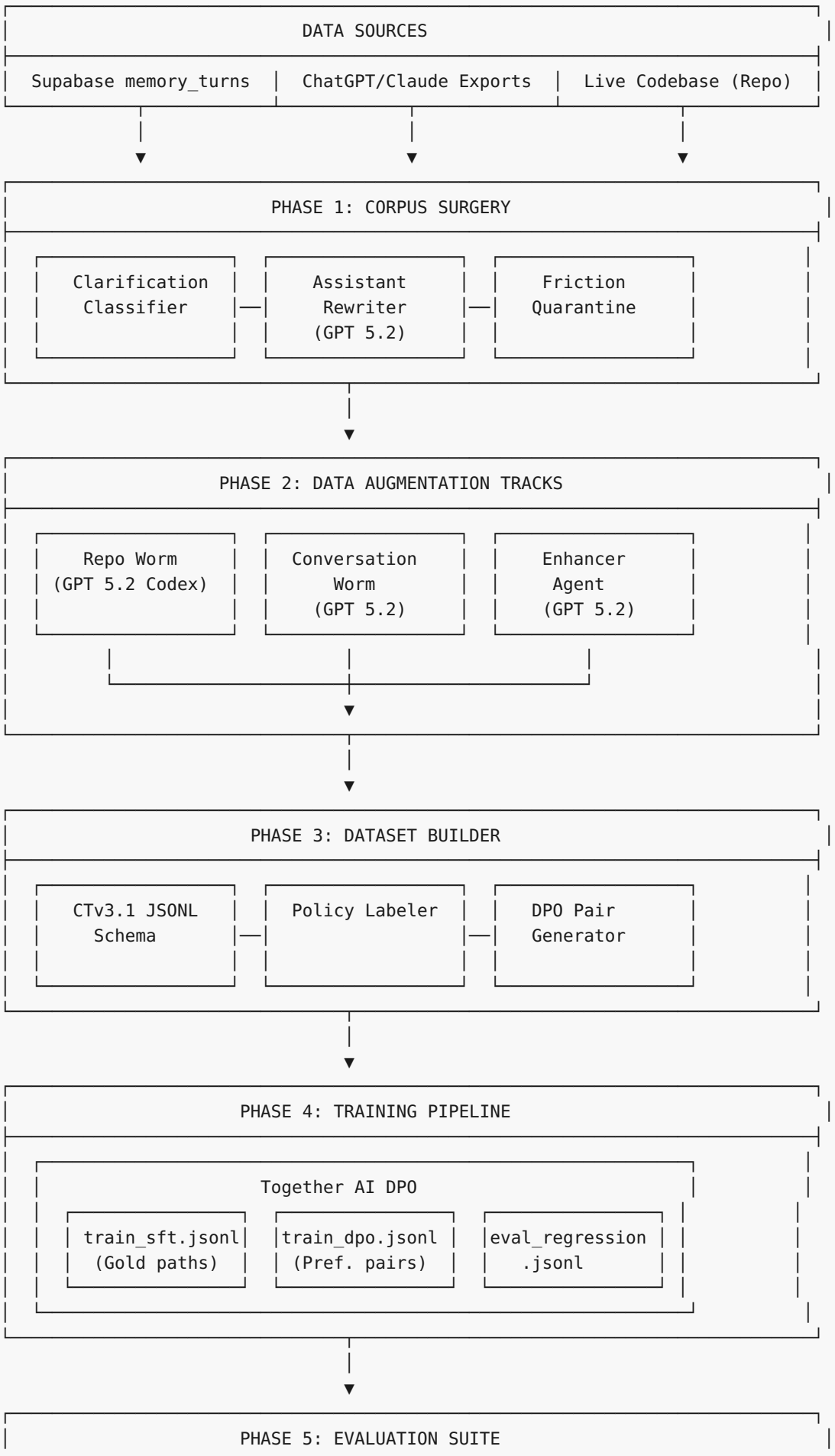
- Required input is genuinely missing
- Ambiguity would change the output materially
- Safety or legal constraints apply

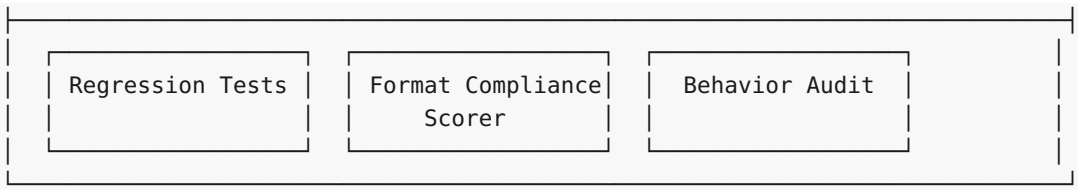
1.3.2. Unjustified Clarification Criteria

- Asking for format preference when one is acceptable
- Confirming before obvious transformations
- Offering options when a default is reasonable

---

2. Architecture Diagram


DATA SOURCES
Supabase memory_turns
ChatGPT/Claude Exports
Live Codebase (Repo)
PHASE 1: CORPUS SURGERY
Clarification Classifier
Assistant Rewriter (GPT 5.2)
Friction Quarantine
PHASE 2: DATA AUGMENTATION TRACKS
Repo Worm (GPT 5.2 Codex)
Conversation Worm (GPT 5.2)
Enhancer Agent (GPT 5.2)
PHASE 3: DATASET BUILDER
CTv3.1 JSONL Schema
Policy Labeler
DPO Pair Generator
PHASE 4: TRAINING PIPELINE
Together AI DPO
train_sft.jsonl (Gold paths)
train_dpo.jsonl (Pref. pairs)
eval_regression .jsonl
PHASE 5: EVALUATION SUITE



3. Quick Reference Table

Phase	Document	Key Components	Implementation Files
0	00_OVERVIEW.md	Goals, Architecture, Glossary	-
1	01_CORPUS_SURGERY.md	Classifier, Rewriter, Quarantine	`corpus_surgery/*.py`
2A	02_REPO_WORM.md	Code Graph, Task Generation	`worms/repo_worm.py`
2B	03_CONVERSATION_WORM.md	Topology Branching, Repair Elimination	`worms/conversation_worm.py`
2C	04_ENHANCER_AGENT.md	Canonicalization, Completion	`worms/enhancer_agent.py`
3	05_DATASET_BUILDER.md	CTv3.1 Schema, Labeling, DPO Pairs	`dataset/*.py`
4	06_TRAINING_PIPELINE.md	Together AI, SFT, DPO	`pipeline.py`
5	07_EVALUATION_SUITE.md	Regression Tests, Metrics	`eval/*.py`
6	08_API_INTEGRATION.md	OpenAI GPT 5.2 / Codex Setup	`api/*.py`

4. Glossary of Terms

4.1. directive_completeness

A scalar value in the range [0.0, 1.0] that measures how complete and unambiguous a user's directive is.

4.1.1. Computation Rules

- Start at 0.0
- Add 0.35 if imperative verb present ("rewrite", "generate", "implement", "extract", "return")
- Add 0.25 if output format specified ("in JSON", "as CSV", "don't omit", "exact rewrite")
- Add 0.20 if all required inputs are present (text, file path, constraints)
- Subtract 0.40 if required input is missing
- Subtract 0.20 if ambiguity changes output materially
- Clamp to [0.0, 1.0]

4.1.2. Thresholds

- > = 0.7: High completeness → `no_questions` policy
- 0.4 - 0.7: Medium completeness → `questions_if_required` policy
- < 0.4: Low completeness → `questions_allowed` policy

4.2. question_policy

An enum that governs whether the assistant may ask questions.

4.2.1. Values

- `no_questions`: Execute immediately, do not ask permission
- `questions_if_required`: Ask only if correctness is blocked
- `questions_allowed`: Open-ended brainstorming, questions permitted

4.2.2. Policy Enforcement

- Classifier tags each turn with appropriate policy
- Rewriter enforces policy during augmentation
- Evaluator checks policy compliance

4.3. stall_score / exec_score / blocked_score

Three integer scores used by the Clarification Classifier.

4.3.1. stall_score

Measures permission-seeking behavior in assistant messages. - +3: Strong permission phrases ("would you like me to", "should i") - +2: Option-dumping phrases ("here are a few options") - +1: Clarification preambles ("i need more information") - +1: Ends with question mark

4.3.2. exec_score

Measures whether assistant actually executed despite asking. - +1: Contains code block - +1: Contains unified diff markers - +1: Contains JSON object - +1: Contains "here is" + substantial content - +1: Contains numbered steps >= 3 - +2: Complete artifact matching format constraint

4.3.3. blocked_score

Measures whether clarification is genuinely required. - Start at 0 if directive_completeness >= 0.7 - +3: Required input genuinely missing - +2: Ambiguous target object - -1: Format specified and feasible - -2: User explicitly asked "choose between"

4.4. Additional Terms

Term	Definition
Gold Trajectory	Conversation path with high quality, minimal friction
Friction Trajectory	Conversation path where user corrected the model
Assumption Protocol	State assumptions as declarations, then proceed
Provider-isms	Phrases like "As an AI language model..."
Control-Repair	User message correcting model behavior

5. Success Criteria

5.1. Quantitative Metrics

Metric	Target	Measurement Method
Clarification Classifier Accuracy	>= 90%	Labeled test set (500 samples)
Unjustified Questions on High-Directive Prompts	0%	Regression suite (100 cases)
Format Compliance Rate	>= 95%	Format-specific test set
DPO Training Loss	< V2 baseline	Together AI metrics
Regression Suite Pass Rate	100%	Automated eval

5.2. Qualitative Criteria

5.2.1. Behavior Audit

- Model executes immediately on clear directives
- Model states assumptions without asking
- Model produces complete artifacts when requested
- Model preserves content when told "don't omit"

5.2.2. User Experience

- No "Would you like me to...?" on directive prompts
- No "Before I proceed..." stalling
- No option-dumping without execution
- Appropriate questions only when genuinely blocked

5.3. Validation Process

1. **Unit Tests**: Each component has comprehensive tests
2. **Integration Tests**: End-to-end pipeline verification
3. **Regression Suite**: 100+ cases from historical annoyances
4. **A/B Evaluation**: V3 vs V2 on held-out prompts
5. **Human Audit**: Manual review of 50 random outputs

---

6. Implementation Files Structure

```
rag_plusplus/ml/cognitivetwin_v3/
├── __init__.py
├── schema.py                          # CTv3.1 JSONL schema dataclasses
├── pipeline.py                        # V3 orchestration pipeline
│
├── corpus_surgery/
│   ├── __init__.py
│   ├── classifier.py                  # Clarification classifier
│   ├── rewriter.py                    # GPT 5.2 assistant rewriter
│   └── quarantine.py                  # Friction trajectory handler
│
├── worms/
│   ├── __init__.py
│   ├── repo_worm.py                   # Codebase traversal (GPT 5.2 Codex)
│   ├── conversation_worm.py           # Topology-consistent branching
│   └── enhancer_agent.py              # Canonicalization and completion
│
├── dataset/
│   ├── __init__.py
│   ├── labeler.py                     # Policy label computation
│   ├── pair_generator.py              # DPO pair generation
│   └── exporter.py                    # Export to train/dpo/eval splits
│
├── eval/
│   ├── __init__.py
│   ├── regression_suite.py            # Regression test framework
│   └── metrics.py                     # Evaluation metrics
│
└── api/
    ├── __init__.py
    ├── openai_client.py               # GPT 5.2 / Codex client
    └── together_client.py             # Together AI training client
```

7. Dependencies

7.1. External Services

Service	Purpose	Configuration
OpenAI API	GPT 5.2, GPT 5.2 Codex	`OPENAI_API_KEY`
Together AI	DPO Fine-tuning	`TOGETHER_API_KEY`
Supabase	Corpus storage	`SUPABASE_URL`, `SUPABASE_KEY`

7.2. Python Packages

```
openai>=1.0.0
together>=0.3.0
supabase>=2.0.0
networkx>=3.0
pydantic>=2.0
```

7.3. Internal Dependencies

- `rag_plusplus.tpo.pipeline.TPOPipeline` - Path extraction
- `rag_plusplus.service.code_graph.builder.CodeGraphBuilder` - Code analysis
- `rag_plusplus.ml.cognitive.feedback.FeedbackLearner` - Preference learning

---

8. Document Navigation

Document	Purpose	Prerequisites
01_CORPUS_SURGERY.md	Classifier, Rewriter, Quarantine	This overview
02_REPO_WORM.md	Code graph task generation	01_CORPUS_SURGERY
03_CONVERSATION_WORM.md	Topology branching	01_CORPUS_SURGERY
04_ENHANCER_AGENT.md	Canonicalization	01_CORPUS_SURGERY
05_DATASET_BUILDER.md	Schema and labeling	02, 03, 04
06_TRAINING_PIPELINE.md	Together AI training	05_DATASET_BUILDER
07_EVALUATION_SUITE.md	Regression testing	06_TRAINING_PIPELINE
08_API_INTEGRATION.md	OpenAI setup	All phases

Phase 1: Corpus Surgery

> **Purpose**: Remove unjustified clarifications from training data, quarantine friction trajectories for DPO/eval use, and rewrite permission-seeking turns into direct execution.
>
> **Implementation Files**: - `rag_plusplus/ml/cognitivetwin_v3/corpus_surgery/classifier.py` - `rag_plusplus/ml/cognitivetwin_v3/corpus_surgery/rewriter.py` - `rag_plusplus/ml/cognitivetwin_v3/corpus_surgery/quarantine.py`

---

1. Purpose and Goals

1.1. Remove Unjustified Clarifications from Training Data

1.1.1. Problem Statement

- Training data contains assistant turns that ask for permission when not needed
- These turns teach the model that permission-seeking is the default behavior
- Result: V2 model ends responses with "Would you like me to...?"

1.1.2. Solution Approach

- Build a classifier to detect unjustified clarifications
- Either rewrite or remove these turns
- Replace with direct execution responses

1.1.3. Expected Outcome

- SFT training data contains only "execute immediately" patterns
- Model learns that direct execution is the default
- Permission-seeking becomes the exception, not the rule

1.2. Quarantine Friction Trajectories for DPO/Eval Use

1.2.1. Friction Trajectory Definition

- Conversations where user corrected the model
- User messages containing "stop asking", "don't do that", "I said..."

- Back-and-forth where user had to fight for compliance

1.2.2. Quarantine Purpose

- These are NOT used for imitation learning (SFT)
- They ARE used for preference learning (DPO) as negative examples
- They ARE used for regression testing (eval suite)

1.2.3. Quarantine Process

- Tag conversation segments with `is_friction = true`
- Extract the bad assistant turn as `dispreferred`
- Generate an ideal response as `preferred`
- Create eval_case for regression testing

1.3. Rewrite Permission-Seeking Turns into Direct Execution

1.3.1. Rewrite Trigger Conditions

- Classification == "unjustified"
- directive_completeness >= 0.5
- Not already executed (exec_score == 0)

1.3.2. Rewrite Strategy

- Use GPT 5.2 to generate the response that should have been given
- Apply assumption protocol: state assumptions, don't ask
- Produce the artifact that was requested

1.3.3. Rewrite Validation

- Must not end with question mark
- Must not contain permission phrases
- Must contain requested artifact (if applicable)

---

2. Clarification Classifier

2.1. Input Specification

2.1.1. Assistant Message Text

- The full text of the assistant's response
- Includes all content: explanations, code blocks, questions

2.1.2. Preceding User Message (for Context)

- The user message that triggered this response
- Used to compute directive_completeness
- Used to determine if clarification is justified

2.1.3. Conversation Phase ID

- Phase 0: Opening/Introduction
- Phase 1: Context Gathering
- Phase 2: Solution Development
- Phase 3: Refinement
- Phase 4: Synthesis
- Phase 5: Conclusion

2.1.4. Format Constraints from User

- `must_not_omit`: User said "don't omit", "full", "exact", "all"
- `forbid_bullets`: User said "no bullets"
- `require_numbered`: User said "numbered list"
- `must_return_code`: User asked for code
- `must_return_diff`: User asked for diff
- `must_return_json`: User asked for JSON

2.2. Text Normalization

2.2.1. Strip Code Blocks (Preserve Placeholders)

```
import re

def strip_code_blocks(text: str) -> str:
    """Replace code blocks with placeholders for pattern matching."""
    pattern = r"```[\s\S]*?```"
    counter = [0]

    def replacer(match):
        counter[0] += 1
        return f"<CODE_BLOCK_{counter[0]}>"

    return re.sub(pattern, replacer, text)
```

2.2.2. Strip Quoted User Text

```
def strip_quoted_text(text: str) -> str:
    """Remove quoted text (user content being referenced)."""
    # Remove > quoted lines
    lines = text.split('\n')
    lines = [l for l in lines if not l.strip().startswith('>')]

    # Remove "you said" style quotes
    pattern = r'"[^"]{50,}"'  # Long quotes are likely user content
    text = '\n'.join(lines)
    return re.sub(pattern, '<QUOTED_TEXT>', text)
```

2.2.3. Lowercase for Pattern Matching

```
def normalize_for_matching(text: str) -> str:
    """Normalize text for phrase matching."""
    text = strip_code_blocks(text)
    text = strip_quoted_text(text)
    return text.lower()
```

2.2.4. Preserve Punctuation for ? Detection

- Keep question marks in normalized text
- Track position of question marks (especially at end)
- Detect rhetorical vs. genuine questions

2.3. Stall Score Computation

2.3.1. Strong Permission Phrases (+3 each)

Phrase	Score	Example
"would you like me to"	+3	"Would you like me to implement this?"
"do you want me to"	+3	"Do you want me to refactor this?"
"should i"	+3	"Should I use TypeScript or JavaScript?"
"shall i"	+3	"Shall I proceed with this approach?"
"can i proceed"	+3	"Can I proceed with the refactoring?"
"before i proceed"	+3	"Before I proceed, I need to ask..."
"can you confirm"	+3	"Can you confirm this is correct?"
"please confirm"	+3	"Please confirm before I continue."
"let me know if you want"	+3	"Let me know if you want me to..."
"tell me if you want"	+3	"Tell me if you want changes."
"is that okay"	+3	"Is that okay with you?"
"does that work"	+3	"Does that work for you?"
"sound good"	+3	"Does that sound good?"
"would you prefer"	+3	"Would you prefer option A or B?"
"should we"	+3	"Should we use this approach?"

```
STRONG_PERMISSION_PHRASES = [
    "would you like me to",
    "do you want me to",
    "should i",
    "shall i",
    "can i proceed",
    "before i proceed",
    "can you confirm",
    "please confirm",
    "let me know if you want",
    "tell me if you want",
    "is that okay",
    "does that work",
    "sound good",
    "would you prefer",
    "should we",
]
STRONG_PERMISSION_SCORE = 3
```

2.3.2. Option-Dumping Phrases (+2 each)

Phrase	Score	Example
"i can do x or y"	+2	"I can implement this in Python or JavaScript."
"here are a few options"	+2	"Here are a few options to consider:"
"which approach do you want"	+2	"Which approach do you want me to take?"
"pick one of the following"	+2	"Pick one of the following options:"
"choose between"	+2	"You can choose between A and B."
"a few ways to"	+2	"There are a few ways to do this:"
"several approaches"	+2	"There are several approaches:"
"multiple options"	+2	"We have multiple options here."

```
OPTION_DUMPING_PHRASES = [
    "i can do",
    "here are a few options",
    "here are some options",
    "which approach do you want",
    "pick one of the following",
    "choose between",
    "a few ways to",
    "several approaches",
    "multiple options",
    "we could either",
]
OPTION_DUMPING_SCORE = 2
```

2.3.3. Clarification Preambles (+1 each)

Phrase	Score	Example
"i need a bit more information"	+1	"I need a bit more information to help you."
"i'll need more context"	+1	"I'll need more context about your setup."
"to help you better"	+1	"To help you better, could you clarify..."
"could you provide"	+1	"Could you provide more details?"
"what exactly do you mean"	+1	"What exactly do you mean by that?"
"could you clarify"	+1	"Could you clarify what you want?"
"to make sure i understand"	+1	"To make sure I understand correctly..."
"just to clarify"	+1	"Just to clarify, you want..."

```
CLARIFICATION_PREAMBLES = [
    "i need a bit more information",
    "i'll need more context",
    "to help you better",
    "could you provide",
    "what exactly do you mean",
    "could you clarify",
    "to make sure i understand",
    "just to clarify",
    "can you tell me more",
    "what do you mean by",
]
CLARIFICATION_PREAMBLE_SCORE = 1
```

2.3.4. End-of-Message Question Mark (+1)

```
def ends_with_question(text: str) -> bool:
    """Check if message ends with a question."""
    # Strip whitespace and code blocks at end
    text = text.rstrip()

    # Check last non-whitespace character
    if text.endswith('?'):
        return True

    # Check if last sentence is a question
    sentences = re.split(r'[.!?]', text)
    last_sentence = sentences[-1].strip() if sentences else ""

    # Detect question words at start of last sentence
    question_starters = ['what', 'how', 'when', 'where', 'why', 'which',
                         'would', 'should', 'could', 'can', 'do', 'does',
                         'is', 'are', 'will']

    return any(last_sentence.lower().startswith(q) for q in question_starters)

END_QUESTION_SCORE = 1
```

2.3.5. Complete Stall Score Computation

```
def compute_stall_score(text: str) -> int:
    """Compute total stall score for assistant message."""
    normalized = normalize_for_matching(text)
    score = 0

    # Strong permission phrases
    for phrase in STRONG_PERMISSION_PHRASES:
        if phrase in normalized:
            score += STRONG_PERMISSION_SCORE

    # Option-dumping phrases
    for phrase in OPTION_DUMPING_PHRASES:
        if phrase in normalized:
            score += OPTION_DUMPING_SCORE

    # Clarification preambles
    for phrase in CLARIFICATION_PREAMBLES:
        if phrase in normalized:
            score += CLARIFICATION_PREAMBLE_SCORE

    # End-of-message question
    if ends_with_question(text):
        score += END_QUESTION_SCORE

    return score
```

2.4. Exec Score Computation

2.4.1. Code Block Present (+1)

```
def has_code_block(text: str) -> bool:
    """Check if message contains a code block."""
    return bool(re.search(r"```[\s\S]*?```", text))

CODE_BLOCK_SCORE = 1
```

2.4.2. Unified Diff Markers (+1)

```
def has_diff_markers(text: str) -> bool:
    """Check if message contains unified diff markers."""
    diff_patterns = [
        r"^---\s+",      # File header
        r"^\+\+\+\s+",   # File header
        r"^@@.*@@",      # Hunk header
    ]
    for pattern in diff_patterns:
        if re.search(pattern, text, re.MULTILINE):
            return True
    return False

DIFF_MARKER_SCORE = 1
```

2.4.3. JSON Object with Keys (+1)

```
def has_json_object(text: str) -> bool:
    """Check if message contains a JSON-like object."""
    # Look for { followed by "key": pattern
    pattern = r'\{[^}]*"[^"]+"\s*:'
    return bool(re.search(pattern, text))

JSON_OBJECT_SCORE = 1
```

2.4.4. "Here is" + Substantial Content (+1)

```
def has_here_is_content(text: str) -> bool:
    """Check if message contains 'here is' followed by substantial content."""
    pattern = r"here is[^.]*[.:](.{100,})"  # At least 100 chars after
    return bool(re.search(pattern, text.lower(), re.DOTALL))

HERE_IS_SCORE = 1
```

2.4.5. Numbered Steps >= 3 Items (+1)

```
def has_numbered_steps(text: str) -> bool:
    """Check if message contains at least 3 numbered steps."""
    # Look for patterns like "1.", "2.", "3." or "1)", "2)", "3)"
    numbered_pattern = r"^\s*\d+[.)]\s+"
    matches = re.findall(numbered_pattern, text, re.MULTILINE)
    return len(matches) >= 3

NUMBERED_STEPS_SCORE = 1
```

2.4.6. Complete Artifact Matching Format Constraint (+2)

```
def has_complete_artifact(text: str, format_constraints: dict) -> bool:
    """Check if message contains complete artifact matching constraints."""
    if format_constraints.get("must_return_json"):
        # Check for valid JSON block
        try:
            import json
            # Extract JSON from code block
            match = re.search(r"```json\s*([\s\S]*?)```", text)
            if match:
                json.loads(match.group(1))
                return True
        except:
            pass

    if format_constraints.get("must_return_diff"):
        if has_diff_markers(text):
            return True

    if format_constraints.get("must_return_code"):
        if has_code_block(text):
            return True

    return False

COMPLETE_ARTIFACT_SCORE = 2
```

2.4.7. Complete Exec Score Computation

```
def compute_exec_score(text: str, format_constraints: dict = None) -> int:
    """Compute total exec score for assistant message."""
    format_constraints = format_constraints or {}
    score = 0

    if has_code_block(text):
        score += CODE_BLOCK_SCORE

    if has_diff_markers(text):
        score += DIFF_MARKER_SCORE

    if has_json_object(text):
        score += JSON_OBJECT_SCORE

    if has_here_is_content(text):
        score += HERE_IS_SCORE

    if has_numbered_steps(text):
        score += NUMBERED_STEPS_SCORE

    if has_complete_artifact(text, format_constraints):
        score += COMPLETE_ARTIFACT_SCORE

    return score
```

2.5. Blocked Score Computation

2.5.1. directive_completeness >= 0.7 → Start at 0

```
def compute_initial_blocked_score(directive_completeness: float) -> int:
    """Compute initial blocked score based on directive completeness."""
    if directive_completeness >= 0.7:
        return 0
    elif directive_completeness >= 0.4:
        return 1
    else:
        return 2
```

2.5.2. Missing Required Input → +3

```
def check_missing_input(user_message: str, assistant_message: str) -> bool:
    """Check if required input is genuinely missing."""
    # User asked for transformation but didn't provide input
    transformation_words = ["enhance", "refactor", "rewrite", "transform",
                            "convert", "translate", "summarize"]

    user_lower = user_message.lower()

    for word in transformation_words:
        if word in user_lower:
            # Check if any substantial content is provided
            # (code block, long text, file path)
            has_code = bool(re.search(r"```[\s\S]*?```", user_message))
            has_long_text = len(user_message) > 200
            has_file_path = bool(re.search(r"[/\\][\w./\\]+\.\w+", user_message))

            if not (has_code or has_long_text or has_file_path):
                return True

    return False

MISSING_INPUT_SCORE = 3
```

2.5.3. Ambiguous Target → +2

```
def check_ambiguous_target(user_message: str) -> bool:
    """Check if target object is ambiguous."""
    ambiguous_patterns = [
        r"(this|that|it)\s+(function|code|file|module)",  # Vague reference
        r"the\s+(above|below|previous)",                   # Positional reference
        r"fix\s+(the|this|that)\s+bug",                    # Unspecified bug
    ]

    user_lower = user_message.lower()

    for pattern in ambiguous_patterns:
        if re.search(pattern, user_lower):
            # Only ambiguous if no code block or file path provided
            has_context = bool(re.search(r"```[\s\S]*?```", user_message))
            if not has_context:
                return True

    return False

AMBIGUOUS_TARGET_SCORE = 2
```

2.5.4. Format Specified and Feasible → -1

```
def check_format_specified(user_message: str) -> bool:
    """Check if output format is specified."""
    format_indicators = [
        r"in json",
        r"as json",
        r"return json",
        r"as csv",
        r"in csv",
        r"as markdown",
        r"in markdown",
        r"don't omit",
        r"exact rewrite",
        r"no bullets",
        r"numbered list",
    ]

    user_lower = user_message.lower()

    for pattern in format_indicators:
        if re.search(pattern, user_lower):
            return True

    return False

FORMAT_SPECIFIED_BONUS = -1
```

2.5.5. User Asked "Choose Between" → -2

```
def check_user_asked_options(user_message: str) -> bool:
    """Check if user explicitly asked for options."""
    option_patterns = [
        r"choose between",
        r"pick between",
        r"which (one|option)",
        r"what are (the|my) options",
        r"give me options",
        r"list (the|some) options",
    ]

    user_lower = user_message.lower()

    for pattern in option_patterns:
        if re.search(pattern, user_lower):
            return True

    return False

USER_ASKED_OPTIONS_BONUS = -2
```

2.5.6. Complete Blocked Score Computation

```
def compute_blocked_score(
    user_message: str,
    assistant_message: str,
    directive_completeness: float
) -> int:
    """Compute total blocked score."""
    score = compute_initial_blocked_score(directive_completeness)

    if check_missing_input(user_message, assistant_message):
        score += MISSING_INPUT_SCORE

    if check_ambiguous_target(user_message):
        score += AMBIGUOUS_TARGET_SCORE

    if check_format_specified(user_message):
        score += FORMAT_SPECIFIED_BONUS

    if check_user_asked_options(user_message):
        score += USER_ASKED_OPTIONS_BONUS

    return max(0, score)  # Clamp to non-negative
```

2.6. Classification Logic

2.6.1. Unjustified Classification Rule

```
def is_unjustified(stall_score: int, blocked_score: int, exec_score: int,
                   text: str, directive_completeness: float) -> bool:
    """Determine if clarification is unjustified."""
    # Primary rule
    if stall_score >= 3 and blocked_score <= 1 and exec_score == 0:
        return True

    # Secondary rule: strong permission phrase + ends with ? + high completeness
    normalized = normalize_for_matching(text)
    has_strong_permission = any(p in normalized for p in STRONG_PERMISSION_PHRASES)

    if (ends_with_question(text) and
        has_strong_permission and
        directive_completeness >= 0.7):
        return True

    return False
```

2.6.2. Justified Classification Rule

```
def is_justified(stall_score: int, blocked_score: int,
                 text: str, question_policy: str) -> bool:
    """Determine if clarification is justified."""
    # Blocked score indicates genuine need
    if blocked_score >= 3:
        return True

    # Question policy explicitly allows
    if question_policy == "questions_allowed":
        return True

    # Questions if required and genuinely missing info
    if question_policy == "questions_if_required" and blocked_score >= 2:
        return True

    return False
```

2.6.3. Complete Classification

```
from dataclasses import dataclass
from enum import Enum

class ClarificationType(Enum):
    UNJUSTIFIED = "unjustified"
    JUSTIFIED = "justified"
    NEUTRAL = "neutral"

@dataclass
class ClassificationResult:
    classification: ClarificationType
    stall_score: int
    exec_score: int
    blocked_score: int
    directive_completeness: float
    reasoning: str

def classify_assistant_turn(
    assistant_message: str,
    user_message: str,
    phase_id: int,
    format_constraints: dict,
    directive_completeness: float,
    question_policy: str = "no_questions"
) -> ClassificationResult:
    """Classify an assistant turn as unjustified, justified, or neutral."""

    stall_score = compute_stall_score(assistant_message)
    exec_score = compute_exec_score(assistant_message, format_constraints)
    blocked_score = compute_blocked_score(
        user_message, assistant_message, directive_completeness
    )

    # Determine classification
    if is_unjustified(stall_score, blocked_score, exec_score,
                      assistant_message, directive_completeness):
        classification = ClarificationType.UNJUSTIFIED
        reasoning = f"stall={stall_score}>=3, blocked={blocked_score}<=1, exec={exec_score}==0"

    elif is_justified(stall_score, blocked_score,
                      assistant_message, question_policy):
        classification = ClarificationType.JUSTIFIED
        reasoning = f"blocked={blocked_score}>=3 or policy={question_policy}"

    else:
        classification = ClarificationType.NEUTRAL
        reasoning = "No unjustified or justified conditions met"

    return ClassificationResult(
        classification=classification,
        stall_score=stall_score,
        exec_score=exec_score,
        blocked_score=blocked_score,
        directive_completeness=directive_completeness,
        reasoning=reasoning
    )
```

3. Assistant Rewriter

3.1. Trigger Conditions

3.1.1. Classification == "unjustified"

```
def should_rewrite(result: ClassificationResult) -> bool:
    """Determine if assistant turn should be rewritten."""
    return result.classification == ClarificationType.UNJUSTIFIED
```

3.1.2. directive_completeness >= 0.5

```
def can_rewrite(result: ClassificationResult) -> bool:
    """Check if we have enough context to rewrite."""
    return result.directive_completeness >= 0.5
```

3.1.3. Not Already Executed (exec_score == 0)

```
def needs_rewrite(result: ClassificationResult) -> bool:
    """Check if rewrite is needed (didn't already execute)."""
    return result.exec_score == 0
```

3.2. GPT 5.2 Integration

3.2.1. System Prompt Specification

```
REWRITER_SYSTEM_PROMPT = """You are a response rewriter for CognitiveTwin V3.

Your task is to rewrite assistant responses that asked for unnecessary permission
into responses that execute immediately.

RULES:
1. NEVER ask questions when the directive is clear
2. NEVER use phrases like "Would you like me to...", "Should I...", "Can you confirm..."
3. If assumptions are needed, STATE them as declarations, then proceed
4. ALWAYS produce the artifact that was requested
5. Keep the same technical content, just remove permission-seeking

ASSUMPTION PROTOCOL:
- If something is unknown but non-blocking: choose a reasonable default
- State assumptions in ONE line at the start: "Assumptions: ..."
- Then proceed with the implementation/answer
- NO question marks after assumptions

OUTPUT FORMAT:
Return ONLY the rewritten assistant response. No explanations.
"""
```

3.2.2. Context Window Construction

```
def build_rewrite_context(
    assistant_message: str,
    user_message: str,
    conversation_history: list[dict],
    format_constraints: dict
) -> list[dict]:
    """Build context for rewrite request."""

    # Include relevant history (last 3 turns max)
    history_context = conversation_history[-6:] if conversation_history else []

    # Build the rewrite request
    messages = [
        {"role": "system", "content": REWRITER_SYSTEM_PROMPT},
    ]

    # Add history
    for turn in history_context:
        messages.append(turn)

    # Add the problematic turn
    messages.append({
        "role": "user",
        "content": f"""ORIGINAL USER MESSAGE:
{user_message}

PROBLEMATIC ASSISTANT RESPONSE (asks permission when shouldn't):
{assistant_message}

FORMAT CONSTRAINTS: {json.dumps(format_constraints)}

Rewrite this response to execute immediately without asking permission.
Apply the assumption protocol if needed."""
    })

    return messages
```

3.2.3. Temperature Setting (0.3 for Determinism)

```
from openai import OpenAI

async def rewrite_assistant_turn(
    assistant_message: str,
    user_message: str,
    conversation_history: list[dict],
    format_constraints: dict
) -> str:
    """Rewrite an assistant turn using GPT 5.2."""

    client = OpenAI()

    messages = build_rewrite_context(
        assistant_message,
        user_message,
        conversation_history,
        format_constraints
    )

    response = await client.chat.completions.create(
        model="gpt-5.2",
        messages=messages,
        temperature=0.3,  # Low temperature for determinism
        max_tokens=4096,
    )

    return response.choices[0].message.content
```

3.3. Rewrite Rules

3.3.1. Artifact Production: If User Asked → Produce

```
ARTIFACT_TRIGGERS = {
    "implement": "code",
    "create": "code",
    "write": "code",
    "generate": "code",
    "build": "code",
    "refactor": "diff",
    "rewrite": "content",
    "transform": "content",
    "convert": "content",
    "analyze": "analysis",
    "explain": "explanation",
    "summarize": "summary",
    "extract": "data",
    "parse": "data",
}

def detect_required_artifact(user_message: str) -> str | None:
    """Detect what artifact the user expects."""
    user_lower = user_message.lower()

    for trigger, artifact_type in ARTIFACT_TRIGGERS.items():
        if trigger in user_lower:
            return artifact_type

    return None
```

3.3.2. Transformation: If User Asked → Transform

```
def requires_transformation(user_message: str) -> bool:
    """Check if user asked for a transformation."""
    transformation_words = [
        "refactor", "rewrite", "transform", "convert",
        "translate", "migrate", "update", "modify"
    ]

    user_lower = user_message.lower()
    return any(word in user_lower for word in transformation_words)
```

3.3.3. Analysis: If User Asked → Analyze

```
def requires_analysis(user_message: str) -> bool:
    """Check if user asked for analysis."""
    analysis_words = [
        "analyze", "explain", "describe", "evaluate",
        "assess", "review", "audit", "examine"
    ]

    user_lower = user_message.lower()
    return any(word in user_lower for word in analysis_words)
```

3.3.4. Assumption Protocol: State Assumptions, No Questions

````
ASSUMPTION_PROTOCOL_PROMPT = """
When rewriting, if you need to make assumptions:

1. State them in ONE line at the very start:
   "Assumptions: [assumption 1], [assumption 2]"

2. Then proceed with the full response

3. NEVER end with a question

4. NEVER ask for confirmation of assumptions

Example:
BAD: "I could implement this in Python or JavaScript. Which would you prefer?"
GOOD: "Assumptions: Using Python 3.11, following PEP 8 style.

Here is the implementation:
```python
def my_function():
    ...
```"
"""
````

3.4. Output Validation

3.4.1. Must Not End with ?

```
def validate_no_end_question(rewritten: str) -> bool:
    """Validate that rewritten response doesn't end with question."""
    return not ends_with_question(rewritten)
```

3.4.2. Must Not Contain Permission Phrases

```
def validate_no_permission_phrases(rewritten: str) -> bool:
    """Validate that rewritten response doesn't contain permission phrases."""
    normalized = normalize_for_matching(rewritten)

    all_phrases = (STRONG_PERMISSION_PHRASES +
                   OPTION_DUMPING_PHRASES +
                   CLARIFICATION_PREAMBLES)

    return not any(phrase in normalized for phrase in all_phrases)
```

3.4.3. Must Contain Artifact If Directive Requested

```
def validate_artifact_present(
    rewritten: str,
    user_message: str,
    format_constraints: dict
) -> bool:
    """Validate that required artifact is present."""

    required_artifact = detect_required_artifact(user_message)

    if required_artifact == "code":
        return has_code_block(rewritten)

    if required_artifact == "diff":
        return has_diff_markers(rewritten) or has_code_block(rewritten)

    if format_constraints.get("must_return_json"):
        return has_json_object(rewritten)

    # Other artifacts: just check for substantial content
    if required_artifact:
        return len(rewritten) > 100

    return True  # No artifact required
```

3.4.4. Complete Validation

```
@dataclass
class ValidationResult:
    is_valid: bool
    errors: list[str]

def validate_rewrite(
    rewritten: str,
    user_message: str,
    format_constraints: dict
) -> ValidationResult:
    """Validate a rewritten assistant response."""
    errors = []

    if not validate_no_end_question(rewritten):
        errors.append("Response ends with question")

    if not validate_no_permission_phrases(rewritten):
        errors.append("Response contains permission phrases")

    if not validate_artifact_present(rewritten, user_message, format_constraints):
        errors.append("Required artifact not present")

    return ValidationResult(
        is_valid=len(errors) == 0,
        errors=errors
    )
```

---

4. Friction Quarantine

4.1. User Frustration Triggers

4.1.1. Trigger Phrases

Trigger	Description
"stop asking"	User explicitly telling model to stop
"don't do that"	User correcting behavior
"i said"	User repeating instruction (implies model ignored)
"just do it"	User demanding execution
"i challenge you"	User escalating to force compliance
"actually"	User correcting model's interpretation
"no, i meant"	User clarifying after misunderstanding
"that's not what i asked"	User indicating wrong output
"try again"	User requesting redo
"you keep"	User noting repeated bad behavior

```
FRUSTRATION_TRIGGERS = [
    "stop asking",
    "don't ask",
    "don't do that",
    "i said",
    "just do it",
    "i challenge you",
    "actually,",
    "no, i meant",
    "that's not what i asked",
    "try again",
    "you keep",
    "i already told you",
    "as i mentioned",
    "like i said",
    "for the third time",
    "please just",
]

def detect_frustration(user_message: str) -> bool:
    """Detect if user message contains frustration triggers."""
    user_lower = user_message.lower()
    return any(trigger in user_lower for trigger in FRUSTRATION_TRIGGERS)
```

4.2. Quarantine Actions

4.2.1. Mark Conversation Segment

```
@dataclass
class QuarantineMarker:
    conversation_id: str
    start_turn_idx: int
    end_turn_idx: int
    trigger_phrase: str
    bad_assistant_turn: str
    user_correction: str
    is_friction: bool = True

def mark_friction_segment(
    conversation: list[dict],
    frustration_turn_idx: int
) -> QuarantineMarker:
    """Mark a conversation segment as friction."""

    # Find the bad assistant turn (immediately before frustration)
    bad_turn_idx = frustration_turn_idx - 1

    # Find where the friction started (look back for permission-seeking)
    start_idx = bad_turn_idx
    while start_idx > 0:
        prev_turn = conversation[start_idx - 1]
        if prev_turn.get("role") == "assistant":
            result = classify_assistant_turn(prev_turn["content"], ...)
            if result.classification == ClarificationType.UNJUSTIFIED:
                start_idx -= 2  # Include the user turn before
            else:
                break
        else:
            break

    return QuarantineMarker(
        conversation_id=conversation[0].get("conversation_id", "unknown"),
        start_turn_idx=start_idx,
        end_turn_idx=frustration_turn_idx,
        trigger_phrase=detect_trigger_phrase(conversation[frustration_turn_idx]),
        bad_assistant_turn=conversation[bad_turn_idx]["content"],
        user_correction=conversation[frustration_turn_idx]["content"],
    )
```

4.2.2. Extract as DPO Negative Example

```
@dataclass
class DPOPair:
    prompt: str
    preferred: str
    dispreferred: str
    confidence: float
    source: str

def create_dpo_pair_from_quarantine(
    marker: QuarantineMarker,
    conversation: list[dict],
    ideal_response: str
) -> DPOPair:
    """Create a DPO pair from quarantined segment."""

    # Build prompt from context before bad turn
    prompt_turns = conversation[:marker.start_turn_idx]
    prompt = format_conversation_as_prompt(prompt_turns)

    # Dispreferred is the bad assistant turn
    dispreferred = marker.bad_assistant_turn

    # Preferred is the ideal response (generated or corrected)
    preferred = ideal_response

    return DPOPair(
        prompt=prompt,
        preferred=preferred,
        dispreferred=dispreferred,
        confidence=0.9,  # High confidence since user explicitly corrected
        source="friction_quarantine"
    )
```

4.2.3. Create eval_case for Regression

```
@dataclass
class EvalCase:
    record_type: str = "eval_case"
    prompt: str = ""
    expected_behavior: list[str] = None
    disallowed_behaviors: list[str] = None
    reference_answer: str | None = None

def create_eval_case_from_quarantine(
    marker: QuarantineMarker,
    conversation: list[dict]
) -> EvalCase:
    """Create an eval case for regression testing."""

    # Build prompt
    prompt_turns = conversation[:marker.start_turn_idx + 1]  # Include user turn
    prompt = format_conversation_as_prompt(prompt_turns)

    return EvalCase(
        record_type="eval_case",
        prompt=prompt,
        expected_behavior=[
            "Executes immediately without asking permission",
            "Does not end with a question",
            "Produces the requested artifact",
        ],
        disallowed_behaviors=[
            "would you like me to",
            "should i",
            "before i proceed",
            "ends_with_question",
        ],
        reference_answer=None,  # Can be filled by enhancer agent
    )
```

4.2.4. Exclude from SFT Training Set

```
def filter_sft_data(
    records: list[dict],
    quarantine_markers: list[QuarantineMarker]
) -> list[dict]:
    """Filter out quarantined segments from SFT training data."""

    # Build set of quarantined turn IDs
    quarantined_ids = set()
    for marker in quarantine_markers:
        for idx in range(marker.start_turn_idx, marker.end_turn_idx + 1):
            quarantined_ids.add((marker.conversation_id, idx))

    # Filter records
    filtered = []
    for record in records:
        conv_id = record.get("conversation_id")
        turn_idx = record.get("turn_idx")

        if (conv_id, turn_idx) not in quarantined_ids:
            filtered.append(record)

    return filtered
```

4.3. Alternate Branch Generation

4.3.1. Generate Ideal Assistant Response

```
async def generate_ideal_response(
    bad_assistant_turn: str,
    user_message: str,
    conversation_history: list[dict],
    format_constraints: dict
) -> str:
    """Generate the ideal assistant response that should have been given."""

    # Use the rewriter with additional context about what went wrong
    system_prompt = REWRITER_SYSTEM_PROMPT + """

ADDITIONAL CONTEXT:
This assistant turn caused the user to become frustrated and correct the model.
Your task is to generate what the assistant SHOULD have said originally.
Focus on:
1. Immediate execution
2. No permission-seeking
3. Complete artifact delivery
"""

    client = OpenAI()

    messages = [
        {"role": "system", "content": system_prompt},
        *conversation_history[-4:],
        {"role": "user", "content": user_message},
        {"role": "assistant", "content": bad_assistant_turn},
        {"role": "user", "content": f"Generate what the assistant should have said instead. Format
    ]

    response = await client.chat.completions.create(
        model="gpt-5.2",
        messages=messages,
        temperature=0.3,
    )

    return response.choices[0].message.content
```

4.3.2. Create Continuation Without Correction

```
async def generate_gold_continuation(
    ideal_response: str,
    user_message: str,
    conversation_history: list[dict]
) -> list[dict]:
    """Generate a continuation of the conversation assuming ideal behavior."""

    # Build the "repaired" history
    repaired_history = conversation_history.copy()
    repaired_history.append({"role": "user", "content": user_message})
    repaired_history.append({"role": "assistant", "content": ideal_response})

    # Generate a natural follow-up user message
    client = OpenAI()

    response = await client.chat.completions.create(
        model="gpt-5.2",
        messages=[
            {"role": "system", "content": "Generate a natural follow-up user message that continues
            *repaired_history,
        ],
        temperature=0.5,
    )

    follow_up = response.choices[0].message.content
    repaired_history.append({"role": "user", "content": follow_up})

    return repaired_history
```

4.3.3. Use as Gold Trajectory

```
@dataclass
class GoldTrajectory:
    conversation_id: str
    turns: list[dict]
    source: str
    quality_score: float

def create_gold_trajectory(
    repaired_history: list[dict],
    original_conversation_id: str
) -> GoldTrajectory:
    """Create a gold trajectory from repaired conversation."""

    return GoldTrajectory(
        conversation_id=f"{original_conversation_id}_repaired",
        turns=repaired_history,
        source="friction_repair",
        quality_score=0.95,  # High quality since explicitly repaired
    )
```

5. Implementation Files

5.1. classifier.py

```
# rag_plusplus/ml/cognitivetwin_v3/corpus_surgery/classifier.py

"""
Clarification Classifier for CognitiveTwin V3.

Classifies assistant turns as:
- UNJUSTIFIED: Permission-seeking when not needed
- JUSTIFIED: Clarification genuinely required
- NEUTRAL: Neither permission-seeking nor requiring clarification
"""

from dataclasses import dataclass
from enum import Enum
import re
from typing import Optional

# [All the classification code from sections 2.1-2.6]
```

5.2. rewriter.py

```
# rag_plusplus/ml/cognitivetwin_v3/corpus_surgery/rewriter.py

"""
Assistant Rewriter for CognitiveTwin V3.

Uses GPT 5.2 to rewrite unjustified clarifications into direct execution.
"""

from openai import OpenAI
from dataclasses import dataclass
from typing import Optional

# [All the rewriter code from section 3]
```

5.3. quarantine.py

```
# rag_plusplus/ml/cognitivetwin_v3/corpus_surgery/quarantine.py

"""
Friction Quarantine for CognitiveTwin V3.

Detects and quarantines friction trajectories for DPO/eval use.
"""

from dataclasses import dataclass
from typing import Optional

# [All the quarantine code from section 4]
```

---

6. Testing Strategy

6.1. Unit Tests

```
# tests/ml/cognitivetwin_v3/test_classifier.py

def test_stall_score_strong_permission():
    text = "Would you like me to implement this feature?"
    score = compute_stall_score(text)
    assert score >= 3

def test_stall_score_option_dumping():
    text = "Here are a few options: A, B, or C."
    score = compute_stall_score(text)
    assert score >= 2

def test_exec_score_with_code():
    text = "Here is the implementation:\n```python\ndef foo(): pass\n```"
    score = compute_exec_score(text, {})
    assert score >= 2

def test_classification_unjustified():
    result = classify_assistant_turn(
        assistant_message="Should I proceed with this approach?",
        user_message="Implement the login feature.",
        phase_id=2,
        format_constraints={},
        directive_completeness=0.8,
    )
    assert result.classification == ClarificationType.UNJUSTIFIED
```

6.2. Integration Tests

```
# tests/ml/cognitivetwin_v3/test_corpus_surgery.py

async def test_full_pipeline():
    # Load test conversation
    conversation = load_test_conversation()

    # Run classifier
    results = [classify_turn(turn) for turn in conversation]

    # Rewrite unjustified turns
    rewritten = await rewrite_unjustified_turns(conversation, results)

    # Validate rewrites
    for turn in rewritten:
        validation = validate_rewrite(turn)
        assert validation.is_valid
```

Phase 2A: Repo Worm

> **Purpose**: Generate training data from codebase traversal, create code completion tasks, and produce DPO pairs (stalling vs executing).
>
> **Model**: GPT 5.2 Codex (agentic coding model)
>
> **Implementation File**: `rag_plusplus/ml/cognitivetwin_v3/worms/repo_worm.py`

---

1. Purpose

1.1. Generate Training Data from Codebase

1.1.1. Why Codebase-Grounded Data

- Real code context prevents hallucination
- Teaches model actual project patterns and conventions
- Provides verifiable ground truth (code compiles or doesn't)
- Creates tasks that match real developer workflows

1.1.2. Types of Training Data Generated

- Code completion tasks (finish partial implementations)
- Implementation tasks (implement interfaces, fill stubs)
- Refactoring tasks (match patterns, extract modules)
- Test generation tasks (write tests for functions)

1.1.3. Grounding Requirements

- Every claim about codebase must be supported by included snippets
- Worm expands context rather than guessing
- No invented files, symbols, or behaviors
- Assumptions must be testable

1.2. Create Code Completion Tasks

1.2.1. Task Categories

- **TODO Completion**: Finish sections marked with TODO/FIXME

- **Stub Implementation**: Fill in function stubs and placeholders
- **Missing Methods**: Add methods referenced but not implemented
- **Interface Satisfaction**: Implement required protocol/interface methods

1.2.2. Task Difficulty Levels

- **Easy**: Single function, clear interface, existing tests
- **Medium**: Multiple functions, some ambiguity, partial tests
- **Hard**: Cross-module changes, design decisions, no tests

1.2.3. Context Requirements

- Include sufficient file context for understanding
- Include import dependencies
- Include usage examples where available
- Include test expectations if present

1.3. Produce DPO Pairs (Stalling vs Executing)

1.3.1. Dispreferred Patterns

- Asks which approach to take
- Offers multiple options without choosing
- Refuses to proceed without confirmation
- Asks for clarification on obvious details

1.3.2. Preferred Patterns

- Chooses reasonable defaults
- States assumptions briefly
- Produces complete implementation
- Mentions alternatives without asking

1.3.3. Pair Generation Strategy

- For each task, generate both response types
- Use GPT 5.2 Codex for preferred (executing)
- Use template-based generation for dispreferred (stalling)

---

2. Code Graph Construction

2.1. Integration with CodeGraphBuilder

2.1.1. Import Existing Builder

```
from rag_plusplus.service.code_graph.builder import CodeGraphBuilder
from rag_plusplus.service.code_graph.types import (
    CodeNode,
    CodeEdge,
    NodeType,
    EdgeType,
    UnifiedCodeGraph,
)
from rag_plusplus.service.code_graph.coordinates import CodeCoordinateComputer
```

2.1.2. Configure for V3 Task Extraction

```
@dataclass
class RepoWormConfig:
    """Configuration for Repo Worm task extraction."""

    # File filtering
    include_patterns: list[str] = field(default_factory=lambda: [
        "*.py", "*.ts", "*.js", "*.rs", "*.go"
    ])
    exclude_patterns: list[str] = field(default_factory=lambda: [
        "*test*", "*_test*", "*spec*", "node_modules/*", "venv/*"
    ])

    # Task extraction
    max_context_lines: int = 200
    min_function_lines: int = 3
    include_tests_for_context: bool = True

    # Task difficulty
    easy_threshold: int = 20      # Lines of code
    medium_threshold: int = 50
    hard_threshold: int = 100

    # DPO pair generation
    generate_dispreferred: bool = True
    dispreferred_templates: list[str] = field(default_factory=list)
```

2.1.3. Builder Initialization

```
class RepoWorm:
    """Codebase traversal agent for training data generation."""

    def __init__(
        self,
        repo_path: Path,
        config: RepoWormConfig = None,
        openai_client: OpenAI = None,
    ):
        self.repo_path = repo_path
        self.config = config or RepoWormConfig()
        self.openai = openai_client or OpenAI()

        # Initialize code graph builder
        self.graph_builder = CodeGraphBuilder()
        self.graph: UnifiedCodeGraph | None = None

        # Task extraction state
        self.extracted_tasks: list[RepoTask] = []
        self.task_contexts: dict[str, str] = {}

    async def initialize(self):
        """Build the code graph for the repository."""
        self.graph = await self.graph_builder.build_unified(self.repo_path)
        self.coord_computer = CodeCoordinateComputer(
            graph=self.graph,
            anchor=self.repo_path,
        )
```

2.2. Node Scanning

2.2.1. File Paths

```
def scan_file_paths(self) -> list[Path]:
    """Scan repository for relevant file paths."""
    files = []

    for pattern in self.config.include_patterns:
        matches = self.repo_path.rglob(pattern)
        for match in matches:
            # Check exclusions
            excluded = any(
                match.match(excl)
                for excl in self.config.exclude_patterns
            )
            if not excluded:
                files.append(match)

    return files
```

2.2.2. Exported Symbols/Classes

```
import ast

def extract_exports(self, file_path: Path) -> list[dict]:
    """Extract exported symbols from a Python file."""
    exports = []

    try:
        content = file_path.read_text()
        tree = ast.parse(content)

        for node in ast.walk(tree):
            if isinstance(node, ast.ClassDef):
                exports.append({
                    "type": "class",
                    "name": node.name,
                    "line": node.lineno,
                    "end_line": node.end_lineno,
                    "docstring": ast.get_docstring(node),
                    "methods": [
                        m.name for m in node.body
                        if isinstance(m, ast.FunctionDef)
                    ],
                })
            elif isinstance(node, ast.FunctionDef):
                if node.col_offset == 0:  # Top-level function
                    exports.append({
                        "type": "function",
                        "name": node.name,
                        "line": node.lineno,
                        "end_line": node.end_lineno,
                        "docstring": ast.get_docstring(node),
                        "args": [a.arg for a in node.args.args],
                    })
    except:
        pass  # Skip files that can't be parsed

    return exports
```

2.2.3. Function Signatures

```
def extract_signatures(self, file_path: Path) -> list[dict]:
    """Extract function signatures for task generation."""
    signatures = []

    try:
        content = file_path.read_text()
        tree = ast.parse(content)

        for node in ast.walk(tree):
            if isinstance(node, ast.FunctionDef):
                # Build signature string
                args = []
                for arg in node.args.args:
                    arg_str = arg.arg
                    if arg.annotation:
                        arg_str += f": {ast.unparse(arg.annotation)}"
                    args.append(arg_str)

                returns = ""
                if node.returns:
                    returns = f" -> {ast.unparse(node.returns)}"

                signature = f"def {node.name}({', '.join(args)}){returns}"

                # Check if body is stub
                is_stub = self._is_stub_body(node.body)

                signatures.append({
                    "signature": signature,
                    "name": node.name,
                    "is_stub": is_stub,
                    "line": node.lineno,
                    "file": str(file_path),
                })
    except:
        pass

    return signatures

def _is_stub_body(self, body: list) -> bool:
    """Check if function body is a stub."""
    if len(body) == 1:
        stmt = body[0]
        # pass statement
        if isinstance(stmt, ast.Pass):
            return True
        # ... (Ellipsis)
        if isinstance(stmt, ast.Expr) and isinstance(stmt.value, ast.Constant):
            if stmt.value.value is ...:
                return True
        # raise NotImplementedError
        if isinstance(stmt, ast.Raise):
            if isinstance(stmt.exc, ast.Call):
                if isinstance(stmt.exc.func, ast.Name):
                    if stmt.exc.func.id == "NotImplementedError":
                        return True
    return False
```

2.2.4. TODO/FIXME Comments

```
import re

def extract_todos(self, file_path: Path) -> list[dict]:
    """Extract TODO/FIXME comments from file."""
    todos = []

    content = file_path.read_text()
    lines = content.split('\n')

    todo_pattern = re.compile(
        r'#\s*(TODO|FIXME|XXX|HACK|BUG)[\s:]+(.+)',
        re.IGNORECASE
    )

    for i, line in enumerate(lines, 1):
        match = todo_pattern.search(line)
        if match:
            todos.append({
                "type": match.group(1).upper(),
                "text": match.group(2).strip(),
                "line": i,
                "file": str(file_path),
                "context": self._get_context(lines, i, 10),
            })

    return todos

def _get_context(self, lines: list[str], line_num: int, window: int) -> str:
    """Get surrounding context for a line."""
    start = max(0, line_num - window - 1)
    end = min(len(lines), line_num + window)
    return '\n'.join(lines[start:end])
```

2.2.5. Failing Tests

```
import subprocess

def find_failing_tests(self) -> list[dict]:
    """Find failing tests in the repository."""
    failing = []

    try:
        # Run pytest with --collect-only to find tests
        result = subprocess.run(
            ["pytest", "--collect-only", "-q", str(self.repo_path)],
            capture_output=True,
            text=True,
            timeout=60,
        )

        # Run pytest to find failures
        result = subprocess.run(
            ["pytest", str(self.repo_path), "-x", "--tb=no", "-q"],
            capture_output=True,
            text=True,
            timeout=300,
        )

        # Parse failures
        for line in result.stdout.split('\n'):
            if 'FAILED' in line:
                # Extract test name and file
                match = re.match(r'(.+)::(.+) FAILED', line)
                if match:
                    failing.append({
                        "file": match.group(1),
                        "test": match.group(2),
                        "output": result.stdout,
                    })
    except:
        pass

    return failing
```

2.2.6. Stub Functions

```
def find_stubs(self) -> list[dict]:
    """Find all stub functions in the repository."""
    stubs = []

    for file_path in self.scan_file_paths():
        signatures = self.extract_signatures(file_path)
        for sig in signatures:
            if sig["is_stub"]:
                stubs.append({
                    **sig,
                    "context": self._get_file_context(
                        file_path,
                        sig["line"]
                    ),
                })

    return stubs
```

2.2.7. Intention Comments ("should", "plan", "later")

```
def extract_intentions(self, file_path: Path) -> list[dict]:
    """Extract comments indicating future intentions."""
    intentions = []

    content = file_path.read_text()
    lines = content.split('\n')

    intention_patterns = [
        r'#.*\bshould\b',
        r'#.*\bplan\b',
        r'#.*\blater\b',
        r'#.*\beventually\b',
        r'#.*\bfuture\b',
        r'#.*\bneed to\b',
        r'#.*\bwant to\b',
    ]

    combined_pattern = re.compile('|'.join(intention_patterns), re.IGNORECASE)

    for i, line in enumerate(lines, 1):
        if combined_pattern.search(line):
            intentions.append({
                "text": line.strip(),
                "line": i,
                "file": str(file_path),
                "context": self._get_context(lines, i, 5),
            })

    return intentions
```

2.3. Edge Construction

2.3.1. Import Relationships

```
def build_import_edges(self) -> list[CodeEdge]:
    """Build edges for import relationships."""
    edges = []

    for file_path in self.scan_file_paths():
        try:
            content = file_path.read_text()
            tree = ast.parse(content)

            for node in ast.walk(tree):
                if isinstance(node, ast.Import):
                    for alias in node.names:
                        edges.append(CodeEdge(
                            source=str(file_path),
                            target=alias.name,
                            edge_type=EdgeType.IMPORTS,
                        ))
                elif isinstance(node, ast.ImportFrom):
                    if node.module:
                        edges.append(CodeEdge(
                            source=str(file_path),
                            target=node.module,
                            edge_type=EdgeType.IMPORTS,
                        ))
        except:
            pass

    return edges
```

2.3.2. Call Relationships

```
def build_call_edges(self) -> list[CodeEdge]:
    """Build edges for function call relationships."""
    edges = []

    for file_path in self.scan_file_paths():
        try:
            content = file_path.read_text()
            tree = ast.parse(content)

            # Track current function context
            for node in ast.walk(tree):
                if isinstance(node, ast.FunctionDef):
                    caller = f"{file_path}::{node.name}"

                    for child in ast.walk(node):
                        if isinstance(child, ast.Call):
                            if isinstance(child.func, ast.Name):
                                callee = child.func.id
                                edges.append(CodeEdge(
                                    source=caller,
                                    target=callee,
                                    edge_type=EdgeType.CALLS,
                                ))
        except:
            pass

    return edges
```

2.3.3. Reference Relationships

```
def build_reference_edges(self) -> list[CodeEdge]:
    """Build edges for symbol references."""
    edges = []

    # Build symbol table first
    symbols = {}
    for file_path in self.scan_file_paths():
        exports = self.extract_exports(file_path)
        for export in exports:
            symbols[export["name"]] = f"{file_path}::{export['name']}"

    # Find references
    for file_path in self.scan_file_paths():
        try:
            content = file_path.read_text()
            tree = ast.parse(content)

            for node in ast.walk(tree):
                if isinstance(node, ast.Name):
                    if node.id in symbols:
                        edges.append(CodeEdge(
                            source=str(file_path),
                            target=symbols[node.id],
                            edge_type=EdgeType.REFERENCES,
                        ))
        except:
            pass

    return edges
```

2.3.4. TODO Dependencies

```
def build_todo_edges(self) -> list[CodeEdge]:
    """Build edges for TODO dependencies."""
    edges = []

    for file_path in self.scan_file_paths():
        todos = self.extract_todos(file_path)

        for todo in todos:
            # Check if TODO references another symbol
            text = todo["text"].lower()

            # Look for "depends on X", "after X", "needs X"
            dep_patterns = [
                r'depends on (\w+)',
                r'after (\w+)',
                r'needs (\w+)',
                r'requires (\w+)',
                r'blocked by (\w+)',
            ]

            for pattern in dep_patterns:
                match = re.search(pattern, text)
                if match:
                    edges.append(CodeEdge(
                        source=f"{file_path}:{todo['line']}",
                        target=match.group(1),
                        edge_type=EdgeType.TODO_DEPENDENCY,
                    ))

    return edges
```

---

3. Task Generation

3.1. Task Types

3.1.1. Implementation Tasks

```
@dataclass
class ImplementationTask:
    """Task to implement a function or class."""

    task_type: str = "implementation"
    task_id: str = ""

    # What to implement
    target_signature: str = ""
    target_file: str = ""
    target_line: int = 0

    # Context
    interface_definition: str = ""
    usage_examples: list[str] = field(default_factory=list)
    related_implementations: list[str] = field(default_factory=list)

    # Constraints
    must_compile: bool = True
    must_match_patterns: bool = True
    no_new_dependencies: bool = True
    no_questions: bool = True

    # Prompt templates
    prompt_templates: list[str] = field(default_factory=lambda: [
        "Implement {signature} given the interface below.",
        "Complete the function stub for {name}.",
        "Add the missing implementation for {name}.",
    ])

def generate_implementation_tasks(self) -> list[ImplementationTask]:
    """Generate tasks for stub implementations."""
    tasks = []

    stubs = self.find_stubs()

    for stub in stubs:
        # Find related context
        interface = self._find_interface(stub)
        usage = self._find_usage_examples(stub)
        related = self._find_related_implementations(stub)

        task = ImplementationTask(
            task_id=f"impl_{stub['file']}_{stub['name']}",
            target_signature=stub["signature"],
            target_file=stub["file"],
            target_line=stub["line"],
            interface_definition=interface,
            usage_examples=usage,
            related_implementations=related,
        )

        tasks.append(task)

    return tasks
```

3.1.1.1. "Implement X given interface Y"

```
def format_implement_interface_prompt(self, task: ImplementationTask) -> str:
    """Format prompt for interface implementation."""

    return f"""Implement the following function according to its interface:

SIGNATURE:
```python
{task.target_signature}
```

INTERFACE/PROTOCOL:

```
{task.interface_definition}
```

USAGE EXAMPLES: {chr(10).join(f'```python{chr(10)}{ex}{chr(10)}```' for ex in task.usage_examples[:3])}

CONSTRAINTS: - Must compile without errors - Must match existing code patterns - Must not introduce new dependencies - Do NOT ask clarifying questions - make reasonable assumptions

Provide the complete implementation:"""
```

```
##### 3.1.1.2. "Complete function stub"

```python
def format_complete_stub_prompt(self, task: ImplementationTask) -> str:
    """Format prompt for stub completion."""

    return f"""Complete this function stub:

FILE: {task.target_file}
LINE: {task.target_line}

```python
{task.target_signature}
    pass  # TODO: Implement
```

RELATED IMPLEMENTATIONS FOR REFERENCE:
{chr(10).join(f'```python{chr(10)}{impl}{chr(10)}```' for impl in task.related_implementations[:2])}

CONSTRAINTS: - Follow the patterns shown in related implementations - Make reasonable assumptions for any unknowns - Do NOT ask questions - just implement

Provide the complete function body:"""
```
```

````
##### 3.1.1.3. "Add missing method"

```python
def format_add_method_prompt(self, task: ImplementationTask) -> str:
    """Format prompt for adding missing method."""

    return f"""Add the missing method to this class:

CLASS CONTEXT:
```python
{task.interface_definition}
````

MISSING METHOD: {task.target_signature}

The method is referenced but not implemented. Based on the class context and how the method is used, implement it.

USAGE: {chr(10).join(f'`python{chr(10)}{ex}{chr(10)}`' for ex in task.usage_examples[:3])}

CONSTRAINTS: - Must integrate with existing class methods - Follow existing code style - Do NOT ask for clarification

Provide the method implementation:"""

````
#### 3.1.2. Completion Tasks

```python
@dataclass
class CompletionTask:
    """Task to complete TODO sections."""

    task_type: str = "completion"
    task_id: str = ""

    # TODO info
    todo_text: str = ""
    todo_type: str = ""  # TODO, FIXME, etc.
    file: str = ""
    line: int = 0

    # Context
    surrounding_code: str = ""
    function_signature: str = ""
    file_imports: str = ""

    # Constraints
    must_compile: bool = True
    preserve_behavior: bool = True
    no_questions: bool = True
````
````

3.1.2.1. "Complete TODO section"

```
def format_complete_todo_prompt(self, task: CompletionTask) -> str:
    """Format prompt for TODO completion."""

    return f"""Complete this TODO:

FILE: {task.file}
LINE: {task.line}
TODO: {task.todo_text}

SURROUNDING CODE:
```python
{task.surrounding_code}
```

IMPORTS AVAILABLE:

```
{task.file_imports}
```

CONSTRAINTS: - Complete the TODO as specified - Preserve existing behavior - Follow the existing code style - Do NOT ask questions - implement based on the TODO description

Provide the implementation that replaces the TODO:"""
```

```
##### 3.1.2.2. "Finish partial implementation"

```python
def format_finish_partial_prompt(self, task: CompletionTask) -> str:
    """Format prompt for finishing partial implementation."""

    return f"""Finish this partial implementation:

```python
{task.surrounding_code}
```

The code above is incomplete. Based on the function signature and existing logic, complete the implementation.

FUNCTION: {task.function_signature}

CONSTRAINTS: - Complete all unfinished logic - Handle edge cases appropriately - Follow existing patterns - Do NOT ask for clarification

Provide the completed code:"""
```
```

```
#### 3.1.3. Refactoring Tasks

```python
@dataclass
class RefactoringTask:
    """Task to refactor code."""

    task_type: str = "refactoring"
    task_id: str = ""

    # Target
    target_code: str = ""
    target_file: str = ""
    target_lines: tuple[int, int] = (0, 0)

    # Pattern to match
    pattern_example: str = ""
    pattern_file: str = ""

    # Refactoring type
    refactor_type: str = ""  # extract, inline, rename, restructure

    # Constraints
    must_preserve_behavior: bool = True
    must_compile: bool = True
```
```

3.1.3.1. "Refactor to match pattern in Z"

```
def format_refactor_pattern_prompt(self, task: RefactoringTask) -> str:
    """Format prompt for pattern-matching refactor."""

    return f"""Refactor this code to match the pattern used elsewhere:

TARGET CODE (to refactor):
```python
{task.target_code}
```

PATTERN TO MATCH (from {task.pattern_file}):

```
{task.pattern_example}
```

REFACTORING GOAL: Make the target code follow the same patterns and conventions as the example.

CONSTRAINTS: - Preserve the original behavior - Match the style/patterns exactly - Code must compile - Do NOT ask questions

Provide the refactored code:"""
```

````
##### 3.1.3.2. "Extract to module"

```python
def format_extract_module_prompt(self, task: RefactoringTask) -> str:
    """Format prompt for module extraction."""

    return f"""Extract this code into a separate module:

CODE TO EXTRACT:
```python
{task.target_code}
````

CURRENT FILE: {task.target_file}

Create a new module with this code, and update the original file to import from it.

CONSTRAINTS: - New module should be self-contained - Original file should import and use the new module - All tests should still pass - Do NOT ask for module name - choose an appropriate one

Provide: 1. The new module code 2. The updated import statement for the original file"""
````

````
3.1.4. Test Tasks

```python
@dataclass
class TestTask:
    """Task to write tests."""

    task_type: str = "test"
    task_id: str = ""

    # Function to test
    function_signature: str = ""
    function_body: str = ""
    function_file: str = ""

    # Existing tests
    existing_tests: list[str] = field(default_factory=list)

    # Test requirements
    coverage_targets: list[str] = field(default_factory=list)
````

3.1.4.1. "Write tests for function A"

```
def format_write_tests_prompt(self, task: TestTask) -> str:
    """Format prompt for test writing."""

    return f"""Write comprehensive tests for this function:

FUNCTION:
```python
{task.function_signature}
{task.function_body}
```

EXISTING TESTS FOR REFERENCE: {chr(10).join(f'```python{chr(10)}{t}{chr(10)}```' for t in task.existing_tests[:2])}

REQUIREMENTS: - Test normal operation - Test edge cases - Test error conditions - Follow existing test patterns

CONSTRAINTS: - Tests must be runnable with pytest - Use existing fixtures if applicable - Do NOT ask what to test - cover all reasonable cases

Provide the test code:"""
```

```
##### 3.1.4.2. "Add edge case coverage"

```python
def format_edge_case_prompt(self, task: TestTask) -> str:
    """Format prompt for edge case coverage."""

    return f"""Add edge case tests for this function:

FUNCTION:
```python
{task.function_signature}
{task.function_body}
```

EXISTING TESTS: {chr(10).join(f'```python{chr(10)}{t}{chr(10)}```' for t in task.existing_tests)}

The existing tests cover basic cases. Add tests for: - Empty inputs - Boundary values - Invalid inputs - Concurrent access (if applicable) - Resource exhaustion (if applicable)

CONSTRAINTS: - Focus on edge cases not already covered - Follow existing test style - Do NOT ask which cases to add

Provide the additional test code:"""
```
```

```
### 3.2. Context Attachment

#### 3.2.1. Include File Snippets (Bounded)

```python
def get_bounded_context(
    self,
    file_path: Path,
    center_line: int,
    max_lines: int = 200
) -> str:
    """Get bounded context around a specific line."""

    content = file_path.read_text()
    lines = content.split('\n')

    # Calculate window
    half = max_lines // 2
    start = max(0, center_line - half)
    end = min(len(lines), center_line + half)

    # Include line numbers for reference
    numbered = []
    for i, line in enumerate(lines[start:end], start + 1):
        numbered.append(f"{i:4d} | {line}")

    return '\n'.join(numbered)
```

3.2.2. Include Import Context

```
def get_import_context(self, file_path: Path) -> str:
    """Extract import statements from file."""

    content = file_path.read_text()
    lines = content.split('\n')

    imports = []
    for line in lines:
        stripped = line.strip()
        if stripped.startswith('import ') or stripped.startswith('from '):
            imports.append(line)
        elif imports and not stripped:
            # End of import block
            break

    return '\n'.join(imports)
```
```

3.2.3. Include Usage Examples

```
def find_usage_examples(self, symbol_name: str) -> list[str]:
    """Find examples of how a symbol is used."""

    examples = []

    for file_path in self.scan_file_paths():
        content = file_path.read_text()
        lines = content.split('\n')

        for i, line in enumerate(lines):
            if symbol_name in line and not line.strip().startswith('#'):
                # Get surrounding context
                context = self._get_context(lines, i + 1, 3)
                examples.append(context)

                if len(examples) >= 5:
                    return examples

    return examples
```

3.3. Ground Truth Constraints

3.3.1. Must Compile

```
import subprocess
import tempfile

def validate_compiles(self, code: str, file_path: Path) -> bool:
    """Validate that generated code compiles."""

    # Create temp file with the code
    with tempfile.NamedTemporaryFile(
        suffix=file_path.suffix,
        delete=False
    ) as f:
        f.write(code.encode())
        temp_path = f.name

    try:
        if file_path.suffix == '.py':
            result = subprocess.run(
                ['python', '-m', 'py_compile', temp_path],
                capture_output=True,
            )
            return result.returncode == 0

        elif file_path.suffix == '.ts':
            result = subprocess.run(
                ['tsc', '--noEmit', temp_path],
                capture_output=True,
            )
            return result.returncode == 0

        # Add other languages as needed
        return True  # Assume valid if no validator
    finally:
        Path(temp_path).unlink()
```

3.3.2. Must Match Existing Patterns

```
def validate_patterns(self, code: str, reference_code: str) -> bool:
    """Validate that code matches existing patterns."""

    checks = [
        # Indentation style
        self._check_indentation_match(code, reference_code),
        # Naming conventions
        self._check_naming_conventions(code, reference_code),
        # Documentation style
        self._check_docstring_style(code, reference_code),
    ]

    return all(checks)

def _check_indentation_match(self, code: str, reference: str) -> bool:
    """Check if indentation matches reference."""
    # Detect reference indentation (spaces vs tabs, width)
    ref_indent = self._detect_indentation(reference)
    code_indent = self._detect_indentation(code)
    return ref_indent == code_indent
```

3.3.3. Must Not Introduce New Dependencies

```
def validate_no_new_dependencies(
    self,
    code: str,
    existing_imports: set[str]
) -> tuple[bool, list[str]]:
    """Check that no new dependencies are introduced."""

    # Extract imports from generated code
    tree = ast.parse(code)
    new_imports = set()

    for node in ast.walk(tree):
        if isinstance(node, ast.Import):
            for alias in node.names:
                new_imports.add(alias.name.split('.')[0])
        elif isinstance(node, ast.ImportFrom):
            if node.module:
                new_imports.add(node.module.split('.')[0])

    # Check for new dependencies
    added = new_imports - existing_imports

    # Filter out standard library
    stdlib = {'os', 'sys', 're', 'json', 'typing', 'dataclasses',
              'collections', 'functools', 'itertools', 'pathlib'}
    truly_new = added - stdlib

    return len(truly_new) == 0, list(truly_new)
```

3.3.4. Must Not Ask Questions

```
def validate_no_questions(self, response: str) -> bool:
    """Validate that response doesn't ask questions."""

    # Import classifier from corpus surgery
    from .corpus_surgery.classifier import compute_stall_score

    stall_score = compute_stall_score(response)
    return stall_score < 3
```

---

4. GPT 5.2 Codex Integration

4.1. API Configuration

4.1.1. Model: gpt-5.2-codex

```
CODEX_CONFIG = {
    "model": "gpt-5.2-codex",
    "max_tokens": 8192,
    "temperature": 0.2,
}
```

4.1.2. Context Window: Full File Context

```
def prepare_context_window(
    self,
    task: ImplementationTask | CompletionTask | RefactoringTask | TestTask,
    max_context_tokens: int = 16000
) -> str:
    """Prepare context window for Codex."""

    context_parts = []

    # Add file header
    context_parts.append(f"# File: {task.target_file}")

    # Add imports
    imports = self.get_import_context(Path(task.target_file))
    context_parts.append(imports)

    # Add relevant definitions
    if hasattr(task, 'interface_definition') and task.interface_definition:
        context_parts.append(f"\n# Interface/Protocol:\n{task.interface_definition}")

    # Add surrounding code
    if hasattr(task, 'surrounding_code') and task.surrounding_code:
        context_parts.append(f"\n# Surrounding code:\n{task.surrounding_code}")

    # Add examples
    if hasattr(task, 'usage_examples') and task.usage_examples:
        context_parts.append("\n# Usage examples:")
        for ex in task.usage_examples[:3]:
            context_parts.append(ex)

    return '\n'.join(context_parts)
```

4.1.3. Temperature: 0.2 for Determinism

```
async def call_codex(
    self,
    prompt: str,
    context: str,
    task_type: str
) -> str:
    """Call GPT 5.2 Codex with low temperature for deterministic output."""

    response = await self.openai.responses.create(
        model="gpt-5.2-codex",
        input=f"{context}\n\n{prompt}",
        temperature=0.2,  # Low for determinism
        max_tokens=4096,
    )

    return response.output
```

4.2. Prompt Engineering

4.2.1. System Prompt for Code Generation

```
CODEX_SYSTEM_PROMPT = """You are a code generation assistant for CognitiveTwin V3.

RULES:
1. Generate complete, working code that compiles
2. Follow existing patterns and conventions in the codebase
3. Do NOT ask clarifying questions - make reasonable assumptions
4. State assumptions briefly at the start if needed
5. Include proper error handling and edge cases
6. Match the existing code style exactly

OUTPUT FORMAT:
- Provide code in markdown code blocks
- Include brief comments explaining complex logic
- For diffs, use unified diff format

ASSUMPTION PROTOCOL:
If you need to assume something:
- State it in a comment: # Assumption: ...
- Then proceed with implementation
- NO questions
"""
```

4.2.2. Context Injection Format

```
def format_codex_request(
    self,
    task: ImplementationTask,
    context: str
) -> str:
    """Format a complete Codex request."""

    return f"""{CODEX_SYSTEM_PROMPT}

REPOSITORY CONTEXT:
{context}

TASK:
{self.format_implement_interface_prompt(task)}

Provide the implementation:"""
```

4.2.3. Output Format (Unified Diff)

```
def extract_diff_output(self, response: str) -> str | None:
    """Extract unified diff from Codex response."""

    # Look for diff blocks
    diff_pattern = r'```diff\n([\s\S]*?)\n```'
    match = re.search(diff_pattern, response)

    if match:
        return match.group(1)

    # Look for raw diff markers
    if '---' in response and '+++' in response:
        lines = response.split('\n')
        diff_lines = []
        in_diff = False

        for line in lines:
            if line.startswith('---') or line.startswith('+++'):
                in_diff = True
            if in_diff:
                diff_lines.append(line)
                if line.startswith('@@') and len(diff_lines) > 3:
                    # Found complete hunk header
                    pass

        return '\n'.join(diff_lines) if diff_lines else None

    return None

def extract_code_output(self, response: str) -> str | None:
    """Extract code block from Codex response."""

    # Look for python code blocks
    code_pattern = r'```python\n([\s\S]*?)\n```'
    match = re.search(code_pattern, response)

    if match:
        return match.group(1)

    # Look for generic code blocks
    code_pattern = r'```\n([\s\S]*?)\n```'
    match = re.search(code_pattern, response)

    if match:
        return match.group(1)

    return None
```

4.3. Response Parsing

4.3.1. Extract Diff Blocks

```
@dataclass
class ParsedResponse:
    """Parsed response from Codex."""

    code: str | None = None
    diff: str | None = None
    explanation: str | None = None
    assumptions: list[str] = field(default_factory=list)
    is_valid: bool = False
    errors: list[str] = field(default_factory=list)

def parse_codex_response(self, response: str) -> ParsedResponse:
    """Parse a Codex response."""

    result = ParsedResponse()

    # Extract code
    result.code = self.extract_code_output(response)

    # Extract diff
    result.diff = self.extract_diff_output(response)

    # Extract assumptions
    assumption_pattern = r'#\s*Assumption:\s*(.+)'
    result.assumptions = re.findall(assumption_pattern, response)

    # Extract explanation (text outside code blocks)
    explanation_parts = []
    in_code = False
    for line in response.split('\n'):
        if line.startswith('```'):
            in_code = not in_code
        elif not in_code and line.strip():
            explanation_parts.append(line)
    result.explanation = '\n'.join(explanation_parts)

    # Validate
    result.is_valid = bool(result.code or result.diff)

    return result
```

4.3.2. Validate Syntax

```
def validate_syntax(self, code: str, language: str = "python") -> tuple[bool, str]:
    """Validate code syntax."""

    if language == "python":
        try:
            ast.parse(code)
            return True, ""
        except SyntaxError as e:
            return False, str(e)

    elif language == "typescript":
        # Use tsc for validation
        with tempfile.NamedTemporaryFile(suffix='.ts', delete=False) as f:
            f.write(code.encode())
            temp_path = f.name

        try:
            result = subprocess.run(
                ['tsc', '--noEmit', temp_path],
                capture_output=True,
                text=True,
            )
            if result.returncode == 0:
                return True, ""
            return False, result.stderr
        finally:
            Path(temp_path).unlink()

    return True, ""  # Assume valid for unknown languages
```

4.3.3. Check Compilation

```
async def validate_response(
    self,
    response: ParsedResponse,
    task: ImplementationTask
) -> ParsedResponse:
    """Validate a parsed Codex response."""

    errors = []

    # Check syntax
    if response.code:
        is_valid, error = self.validate_syntax(response.code)
        if not is_valid:
            errors.append(f"Syntax error: {error}")

    # Check compilation
    if response.code and not errors:
        compiles = self.validate_compiles(response.code, Path(task.target_file))
        if not compiles:
            errors.append("Code does not compile")

    # Check no new dependencies
    if response.code and not errors:
        existing = self._get_existing_imports(task.target_file)
        no_new, added = self.validate_no_new_dependencies(response.code, existing)
        if not no_new:
            errors.append(f"New dependencies added: {added}")

    # Check no questions
    if not self.validate_no_questions(response.explanation or ""):
        errors.append("Response contains questions")

    response.errors = errors
    response.is_valid = len(errors) == 0

    return response
```

---

5. DPO Pair Generation

5.1. Dispreferred Response Template

5.1.1. Asks for Confirmation

```
DISPREFERRED_CONFIRMATION_TEMPLATES = [
    "I can implement this for you. Would you like me to proceed with {approach}?",
    "Before I implement this, should I use {option_a} or {option_b}?",
    "I'll need to know: do you want this to {option_a} or {option_b}?",
    "Can you confirm that you want me to {action}?",
    "Just to make sure, should I {action}?",
]

def generate_confirmation_dispreferred(
    self,
    task: ImplementationTask
) -> str:
    """Generate a dispreferred response that asks for confirmation."""

    template = random.choice(DISPREFERRED_CONFIRMATION_TEMPLATES)

    # Fill in template
    options = self._generate_options(task)

    return template.format(
        approach=options[0] if options else "the standard approach",
        option_a=options[0] if len(options) > 0 else "option A",
        option_b=options[1] if len(options) > 1 else "option B",
        action=self._summarize_task(task),
    )
```

5.1.2. Offers Options Without Acting

```
DISPREFERRED_OPTIONS_TEMPLATES = [
    """Here are a few ways I could implement this:

1. {option_a}
2. {option_b}
3. {option_c}

Which approach would you prefer?""",

    """I can do this in several ways:

- {option_a}
- {option_b}

Let me know which one you'd like me to use.""",

    """There are multiple approaches here:

{option_a} would be good for {benefit_a}.
{option_b} would be better for {benefit_b}.

What do you think?""",
]

def generate_options_dispreferred(
    self,
    task: ImplementationTask
) -> str:
    """Generate a dispreferred response that offers options."""

    template = random.choice(DISPREFERRED_OPTIONS_TEMPLATES)
    options = self._generate_options(task)

    return template.format(
        option_a=options[0] if len(options) > 0 else "Using standard library",
        option_b=options[1] if len(options) > 1 else "Using a custom implementation",
        option_c=options[2] if len(options) > 2 else "Using a third-party library",
        benefit_a="simplicity",
        benefit_b="performance",
    )
```

5.1.3. Refuses to Proceed

```
DISPREFERRED_REFUSAL_TEMPLATES = [
    "I need more information before I can implement this. Could you provide {missing}?",
    "To proceed, I'll need to know {missing}. Can you clarify?",
    "I can't implement this without knowing {missing}. Please specify.",
    "Before I proceed, could you tell me {missing}?",
]

def generate_refusal_dispreferred(
    self,
    task: ImplementationTask
) -> str:
    """Generate a dispreferred response that refuses to proceed."""

    template = random.choice(DISPREFERRED_REFUSAL_TEMPLATES)

    # Generate fake "missing" info
    fake_missing = [
        "which error handling strategy to use",
        "the exact input format",
        "whether to support async operations",
        "the preferred naming convention",
    ]

    return template.format(missing=random.choice(fake_missing))
```

5.2. Preferred Response Template

5.2.1. Chooses Sane Defaults

```
async def generate_preferred_response(
    self,
    task: ImplementationTask
) -> str:
    """Generate preferred response that executes immediately."""

    # Build prompt that enforces execution
    prompt = f"""{self.format_implement_interface_prompt(task)}

CRITICAL: Execute immediately. Choose reasonable defaults for any unknowns.
State assumptions briefly, then provide the complete implementation.
Do NOT ask questions."""

    context = self.prepare_context_window(task)

    response = await self.call_codex(prompt, context, "implementation")

    return response
```

5.2.2. States Assumptions Briefly

```
def ensure_assumptions_stated(self, response: str) -> str:
    """Ensure assumptions are stated at the start."""

    parsed = self.parse_codex_response(response)

    if parsed.assumptions:
        # Already has assumptions, good
        return response

    # Add assumption header if code makes implicit assumptions
    assumptions = self._infer_assumptions(parsed.code)

    if assumptions:
        assumption_text = "# Assumptions: " + ", ".join(assumptions)

        # Insert at start of code
        if parsed.code:
            return assumption_text + "\n\n" + response

    return response
```

5.2.3. Produces Complete Diff

```
def generate_complete_diff(
    self,
    original_file: str,
    new_code: str,
    file_path: str
) -> str:
    """Generate complete unified diff."""

    import difflib

    original_lines = original_file.split('\n')
    new_lines = new_code.split('\n')

    diff = difflib.unified_diff(
        original_lines,
        new_lines,
        fromfile=f"a/{file_path}",
        tofile=f"b/{file_path}",
        lineterm=""
    )

    return '\n'.join(diff)
```

---

6. Output Schema

6.1. repo_task Record Format

```
@dataclass
class RepoTaskRecord:
    """CTv3.1 record for repo-grounded tasks."""

    schema_version: str = "ctv3.1"
    record_id: str = ""
    record_type: str = "repo_task"

    source: dict = field(default_factory=lambda: {
        "origin": "repo_worm",
        "provider": "gpt-5.2-codex",
        "source_id": "",
        "created_at_utc": "",
    })

    context: dict = field(default_factory=lambda: {
        "domain": "code",
        "language": "en",
        "topology": {},
        "policy": {
            "question_policy": "no_questions",
            "directive_completeness": 0.9,
            "must_not_omit": False,
            "format_constraints": {
                "must_return_code": True,
                "must_return_diff": False,
            }
        }
    })

    input: dict = field(default_factory=lambda: {
        "messages": [],
        "attachments": [],
    })

    target: dict = field(default_factory=lambda: {
        "assistant_content": "",
        "structured": {
            "diff_unified": "",
            "json": {},
        }
    })

    tags: dict = field(default_factory=lambda: {
        "task_type": "implement",
        "prompt_class": "directive",
        "repo_task": {
            "module": "",
            "symbols": [],
            "build_required": True,
            "tests_required": False,
        }
    })

    quality: dict = field(default_factory=lambda: {
        "gold": False,
        "weight": 1.0,
        "review_status": "auto",
```

```
        "failure_modes": [],
    })
```

6.2. Attachment Specification

```
@dataclass
class RepoAttachment:
    """Attachment for repo context."""

    type: str = "repo_context"
    repo: str = ""
    commit_sha: str = ""
    path: str = ""
    span: dict = field(default_factory=lambda: {
        "start_line": 0,
        "end_line": 0,
    })
    content: str = ""

    def to_dict(self) -> dict:
        return {
            "type": self.type,
            "repo": self.repo,
            "commit_sha": self.commit_sha,
            "path": self.path,
            "span": self.span,
            "content": self.content,
        }
```

6.3. Quality Labeling

```
def label_quality(
    self,
    task: ImplementationTask,
    response: ParsedResponse
) -> dict:
    """Label quality for a generated response."""

    quality = {
        "gold": False,
        "weight": 1.0,
        "review_status": "auto",
        "failure_modes": [],
    }

    # Check for failure modes
    if not response.is_valid:
        quality["failure_modes"].extend(response.errors)

    if not self.validate_no_questions(response.explanation or ""):
        quality["failure_modes"].append("asked_permission")

    if response.explanation and response.explanation.rstrip().endswith('?'):
        quality["failure_modes"].append("ended_with_question")

    # Determine if gold
    if not quality["failure_modes"]:
        quality["gold"] = True
        quality["review_status"] = "auto_passed"
    else:
        quality["weight"] = 0.3  # Downweight problematic responses

    return quality
```

---

7. Complete Pipeline

```
class RepoWormPipeline:
    """Complete Repo Worm pipeline for training data generation."""

    async def run(
        self,
        repo_path: Path,
        output_dir: Path,
    ) -> dict:
        """Run the complete pipeline."""

        # Initialize
        worm = RepoWorm(repo_path)
        await worm.initialize()

        # Generate tasks
        impl_tasks = worm.generate_implementation_tasks()
        completion_tasks = worm.generate_completion_tasks()
        refactor_tasks = worm.generate_refactoring_tasks()
        test_tasks = worm.generate_test_tasks()

        all_tasks = impl_tasks + completion_tasks + refactor_tasks + test_tasks

        # Generate responses
        sft_records = []
        dpo_records = []

        for task in all_tasks:
            # Generate preferred response (executes)
            preferred = await worm.generate_preferred_response(task)
            parsed = worm.parse_codex_response(preferred)
            validated = await worm.validate_response(parsed, task)

            if validated.is_valid:
                # Create SFT record
                record = worm.create_sft_record(task, validated)
                sft_records.append(record)

            # Generate dispreferred responses (stalls)
            for template_type in ["confirmation", "options", "refusal"]:
                dispreferred = worm.generate_dispreferred(task, template_type)

                # Create DPO pair
                dpo_pair = worm.create_dpo_pair(
                    task,
                    preferred=validated.code or preferred,
                    dispreferred=dispreferred
                )
                dpo_records.append(dpo_pair)

        # Export
        worm.export_sft(sft_records, output_dir / "repo_sft.jsonl")
        worm.export_dpo(dpo_records, output_dir / "repo_dpo.jsonl")

        return {
            "tasks_generated": len(all_tasks),
            "sft_records": len(sft_records),
            "dpo_pairs": len(dpo_records),
        }
```

Phase 2B: Conversation Worm

> **Purpose**: Generate topology-consistent synthetic dialogues by walking the conversation DAG and producing controlled variants that stay consistent with trajectory coordinates, phase tags, and preferred interaction policy.
>
> **Model**: GPT 5.2 (general augmentation)
>
> **Implementation File**: `rag_plusplus/ml/cognitivetwin_v3/worms/conversation_worm.py`

---

1. Purpose

1.1. Generate Topology-Consistent Synthetic Dialogues

1.1.1. Why Topology Matters

- Conversations have structure (DAG, not linear)
- Branches represent alternative paths (regenerations, edits)
- Phase determines appropriate behavior (questions allowed in opening, not in solution)
- Trajectory coordinates encode semantic-structural position

1.1.2. Consistency Requirements

- New synthetic turns must preserve phase-appropriate behavior
- Trajectory coordinates must be computable for synthetic data
- Generated content must align with existing conversation flow
- Style and technical depth must match surrounding context

1.1.3. Output Types

- Paraphrase variants (same intent, different wording)
- Branch completions (ideal assistant turns for friction points)
- Trajectory-preserving extensions (continue threads coherently)
- Trajectory-contrast pairs (same prompt, different phase)

1.2. Fix Historical Friction Points

1.2.1. Friction Point Identification

- Turns where user corrected the model
- Turns classified as "unjustified clarification"
- Turns that triggered user frustration

1.2.2. Repair Strategy

- Generate ideal assistant response that should have occurred
- Create continuation where correction was never needed
- Remove friction from the gold trajectory

1.2.3. Training Signal

- Friction turns become DPO dispreferred examples
- Repaired turns become SFT gold examples
- The model learns: "behave like me after correction, but without needing correction"

1.3. Produce Phase-Aware Training Data

1.3.1. Phase Definitions

Phase ID	Name	Description	Question Policy
0	Opening	Introduction, context gathering	questions_if_required
1	Context	Deep understanding, clarification	questions_if_required
2	Solution	Active problem-solving	no_questions
3	Refinement	Iterating on solution	no_questions
4	Synthesis	Summarizing, concluding	no_questions
5	Conclusion	Final deliverables	no_questions

1.3.2. Phase-Behavior Mapping

- Opening/Context: Clarifying questions more acceptable
- Solution/Refinement: Execute immediately, no permission-seeking
- Synthesis/Conclusion: Produce deliverables, no options dumping

---

2. TPO Integration

2.1. Path Extraction

2.1.1. Import TPO Pipeline

```
from rag_plusplus.tpo.pipeline import TPOPipeline, TPOConfig
from rag_plusplus.tpo.core.path_extractor import (
    PathExtractor,
    ConversationPath,
    PathNode,
)
from rag_plusplus.tpo.core.coordinates import DLMCoordinate
from rag_plusplus.tpo.core.quality_calculator import PathQualityCalculator
```

2.1.2. Configure for V3 Branch Generation

```
@dataclass
class ConversationWormConfig:
    """Configuration for Conversation Worm."""

    # TPO integration
    tpo_config: TPOConfig = field(default_factory=TPOConfig)

    # Branch generation
    max_branches_per_friction: int = 3
    min_quality_threshold: float = 0.6

    # Phase behavior
    phase_question_policies: dict = field(default_factory=lambda: {
        0: "questions_if_required",  # Opening
        1: "questions_if_required",  # Context
        2: "no_questions",           # Solution
        3: "no_questions",           # Refinement
        4: "no_questions",           # Synthesis
        5: "no_questions",           # Conclusion
    })

    # Generation parameters
    paraphrase_count: int = 2
    extension_max_turns: int = 3

    # Validation
    require_no_questions_above_phase: int = 2
```

2.1.3. Worm Initialization

```
class ConversationWorm:
    """Conversation DAG traversal agent for training data generation."""

    def __init__(
        self,
        supabase_client,
        config: ConversationWormConfig = None,
        openai_client: OpenAI = None,
    ):
        self.client = supabase_client
        self.config = config or ConversationWormConfig()
        self.openai = openai_client or OpenAI()

        # Initialize TPO pipeline
        self.tpo = TPOPipeline(
            supabase_client=supabase_client,
            config=self.config.tpo_config,
        )

        # State
        self.processed_conversations: set[str] = set()
        self.generated_branches: list[SyntheticBranch] = []

    async def process_conversation(
        self,
        conversation_id: str
    ) -> list[SyntheticBranch]:
        """Process a single conversation."""

        # Extract paths using TPO
        results = await self.tpo.process_conversation(conversation_id)

        branches = []

        # Find friction paths
        friction_paths = self._identify_friction_paths(results.paths)

        for path in friction_paths:
            # Generate ideal branch
            ideal = await self._generate_ideal_branch(path)
            branches.append(ideal)

        # Generate paraphrases
        paraphrases = await self._generate_paraphrases(results.paths)
        branches.extend(paraphrases)

        # Generate extensions
        extensions = await self._generate_extensions(results.paths)
        branches.extend(extensions)

        self.generated_branches.extend(branches)
        self.processed_conversations.add(conversation_id)

        return branches
```

2.2. Quality Scoring

2.2.1. Path Quality Computation

```
def score_path_quality(self, path: ConversationPath) -> float:
    """Compute quality score for a path."""

    # Use TPO quality calculator
    calculator = PathQualityCalculator(
        weights=self.config.tpo_config.quality_weights,
    )

    return calculator.calculate(path)
```

2.2.2. Friction Detection

```
def _identify_friction_paths(
    self,
    paths: list[ConversationPath]
) -> list[ConversationPath]:
    """Identify paths that contain friction points."""

    friction_paths = []

    for path in paths:
        has_friction = False

        for node in path.nodes:
            # Check for user frustration
            if node.role == "user":
                from .corpus_surgery.quarantine import detect_frustration
                if detect_frustration(node.content):
                    has_friction = True
                    break

            # Check for unjustified clarification
            if node.role == "assistant":
                from .corpus_surgery.classifier import classify_assistant_turn

                # Get preceding user message
                user_content = self._get_preceding_user(path, node)

                result = classify_assistant_turn(
                    assistant_message=node.content,
                    user_message=user_content,
                    phase_id=self._get_phase(node),
                    format_constraints={},
                    directive_completeness=self._compute_completeness(user_content),
                )

                if result.classification.value == "unjustified":
                    has_friction = True
                    break

        if has_friction:
            friction_paths.append(path)

    return friction_paths
```

2.3. Coordinate Preservation

2.3.1. Computing Coordinates for Synthetic Turns

```
def compute_synthetic_coordinates(
    self,
    parent_node: PathNode,
    synthetic_content: str,
    branch_type: str
) -> DLMCoordinate:
    """Compute 5D coordinates for synthetic turn."""

    parent_coord = DLMCoordinate.from_dict({
        "depth": parent_node.depth,
        "sibling_order": parent_node.sibling_order,
        "homogeneity": parent_node.homogeneity,
        "temporal": parent_node.temporal,
        "complexity": parent_node.complexity,
    })

    # Depth increases by 1
    new_depth = parent_coord.x + 1

    # Sibling order depends on branch type
    if branch_type == "continuation":
        new_sibling = 0  # First response
    elif branch_type == "alternative":
        new_sibling = parent_coord.y + 1  # Additional sibling
    else:
        new_sibling = 0

    # Compute homogeneity (semantic similarity to parent)
    new_homogeneity = self._compute_homogeneity(
        parent_node.content,
        synthetic_content
    )

    # Temporal advances
    new_temporal = min(1.0, parent_coord.t + 0.05)

    # Complexity from content
    new_complexity = self._compute_complexity(synthetic_content)

    return DLMCoordinate(
        x=new_depth,
        y=new_sibling,
        z=new_homogeneity,
        t=new_temporal,
        n=new_complexity,
    )
```

2.3.2. Homogeneity Computation

```
async def _compute_homogeneity(
    self,
    parent_content: str,
    child_content: str
) -> float:
    """Compute semantic similarity between parent and child."""

    # Use embedding similarity
    from rag_plusplus.service.embedding import EmbedderService

    embedder = EmbedderService()

    parent_emb = await embedder.embed(parent_content)
    child_emb = await embedder.embed(child_content)

    # Cosine similarity
    import numpy as np

    similarity = np.dot(parent_emb, child_emb) / (
        np.linalg.norm(parent_emb) * np.linalg.norm(child_emb)
    )

    return float(similarity)
```

---

3. Branch Generation

3.1. Paraphrase Variants

3.1.1. Purpose

- Robustness to how user phrases directives
- Same intent, different wording
- Teaches model to recognize directives in various forms

3.1.2. Generation Strategy

```
PARAPHRASE_SYSTEM_PROMPT = """You are a paraphrase generator for training data.

Generate {count} paraphrases of the user message that:
1. Preserve the exact same intent and meaning
2. Use different wording, sentence structure, or phrasing
3. Maintain the same level of formality
4. Keep any technical terms unchanged

Output format:
PARAPHRASE 1: ...
PARAPHRASE 2: ...
etc.
"""

async def generate_paraphrases(
    self,
    user_message: str,
    count: int = 2
) -> list[str]:
    """Generate paraphrases of user message."""

    response = await self.openai.chat.completions.create(
        model="gpt-5.2",
        messages=[
            {"role": "system", "content": PARAPHRASE_SYSTEM_PROMPT.format(count=count)},
            {"role": "user", "content": f"Generate paraphrases for:\n\n{user_message}"}
        ],
        temperature=0.7,
    )

    # Parse response
    paraphrases = []
    for line in response.choices[0].message.content.split('\n'):
        if line.startswith('PARAPHRASE'):
            _, text = line.split(':', 1)
            paraphrases.append(text.strip())

    return paraphrases[:count]
```

3.1.3. Creating Paraphrase Records

```
@dataclass
class SyntheticBranch:
    """A synthetic branch generated from conversation."""

    branch_type: str  # paraphrase, ideal_response, extension, contrast
    original_conversation_id: str
    parent_node_id: str

    # Content
    messages: list[dict]

    # Coordinates
    coordinates: DLMCoordinate

    # Labels
    phase_id: int
    question_policy: str
    directive_completeness: float

    # Quality
    source: str = "convo_worm"
    is_gold: bool = False

async def _generate_paraphrases(
    self,
    paths: list[ConversationPath]
) -> list[SyntheticBranch]:
    """Generate paraphrase variants for directive prompts."""

    branches = []

    for path in paths:
        for node in path.nodes:
            if node.role != "user":
                continue

            # Check if directive
            completeness = self._compute_completeness(node.content)
            if completeness < 0.5:
                continue

            # Generate paraphrases
            paraphrases = await self.generate_paraphrases(
                node.content,
                self.config.paraphrase_count
            )

            for paraphrase in paraphrases:
                # Get the assistant response that followed
                assistant_response = self._get_following_assistant(path, node)

                if assistant_response:
                    branch = SyntheticBranch(
                        branch_type="paraphrase",
                        original_conversation_id=path.conversation_id,
                        parent_node_id=node.turn_id,
                        messages=[
                            {"role": "user", "content": paraphrase},
```

```
                            {"role": "assistant", "content": assistant_response.content},
                        ],
                        coordinates=self.compute_synthetic_coordinates(
                            node, paraphrase, "alternative"
                        ),
                        phase_id=self._get_phase(node),
                        question_policy=self._get_question_policy(node),
                        directive_completeness=completeness,
                        is_gold=True,  # Paraphrases preserve quality
                    )
                    branches.append(branch)

    return branches
```

3.2. Branch Completions (Ideal Responses)

3.2.1. Purpose

- Fix historical friction points
- Generate what assistant should have said
- Remove permission-seeking from training trajectory

3.2.2. Ideal Response Generation

```
IDEAL_RESPONSE_SYSTEM_PROMPT = """You are generating the ideal assistant response for CognitiveTwin

The original assistant response asked for permission or clarification when it shouldn't have.
Generate what the assistant SHOULD have said instead.

RULES:
1. Execute immediately - do not ask permission
2. If assumptions are needed, state them briefly then proceed
3. Produce the requested artifact/output
4. Do NOT end with a question
5. Match the technical level and style of the conversation

ASSUMPTION PROTOCOL:
- State assumptions as: "Assumptions: [brief list]"
- Then proceed with full response
- NO question marks in assumptions

The original conversation context is provided. Generate only the ideal assistant response.
"""

async def generate_ideal_response(
    self,
    friction_node: PathNode,
    context: list[dict],
    format_constraints: dict
) -> str:
    """Generate ideal response for friction point."""

    # Build context
    context_str = "\n\n".join([
        f"{'User' if m['role'] == 'user' else 'Assistant'}: {m['content'][:500]}"
        for m in context[-4:]
    ])

    response = await self.openai.chat.completions.create(
        model="gpt-5.2",
        messages=[
            {"role": "system", "content": IDEAL_RESPONSE_SYSTEM_PROMPT},
            {"role": "user", "content": f"""CONVERSATION CONTEXT:
{context_str}

USER MESSAGE (that triggered friction):
{context[-1]['content'] if context else 'N/A'}

PROBLEMATIC ASSISTANT RESPONSE (asked permission when shouldn't have):
{friction_node.content[:1000]}

FORMAT CONSTRAINTS: {format_constraints}

Generate the ideal assistant response that executes immediately:"""}
        ],
        temperature=0.3,
    )

    return response.choices[0].message.content
```

3.2.3. Creating Ideal Branch Records

```
async def _generate_ideal_branch(
    self,
    friction_path: ConversationPath
) -> SyntheticBranch:
    """Generate ideal branch for friction path."""

    # Find the friction node
    friction_node = None
    preceding_context = []

    for i, node in enumerate(friction_path.nodes):
        if node.role == "assistant":
            from .corpus_surgery.classifier import classify_assistant_turn

            user_content = self._get_preceding_user_content(friction_path, i)

            result = classify_assistant_turn(
                assistant_message=node.content,
                user_message=user_content,
                phase_id=self._get_phase(node),
                format_constraints={},
                directive_completeness=self._compute_completeness(user_content),
            )

            if result.classification.value == "unjustified":
                friction_node = node
                preceding_context = [
                    {"role": n.role, "content": n.content}
                    for n in friction_path.nodes[:i]
                ]
                break

    if not friction_node:
        return None

    # Generate ideal response
    ideal_content = await self.generate_ideal_response(
        friction_node,
        preceding_context,
        {}
    )

    # Build the repaired messages
    repaired_messages = preceding_context + [
        {"role": "assistant", "content": ideal_content}
    ]

    return SyntheticBranch(
        branch_type="ideal_response",
        original_conversation_id=friction_path.conversation_id,
        parent_node_id=friction_node.turn_id,
        messages=repaired_messages,
        coordinates=self.compute_synthetic_coordinates(
            friction_node, ideal_content, "alternative"
        ),
        phase_id=self._get_phase(friction_node),
        question_policy="no_questions",  # Enforced
        directive_completeness=self._compute_completeness(
```

```
            preceding_context[-1]["content"] if preceding_context else ""
        ),
        is_gold=True,
    )
```

3.3. Trajectory-Preserving Extensions

3.3.1. Purpose

- Teach longer coherence
- Maintain trajectory through extended exchanges
- Show sustained technical analysis

3.3.2. Extension Generation

```
EXTENSION_SYSTEM_PROMPT = """You are extending a conversation for CognitiveTwin V3 training.

Generate a natural continuation of this conversation that:
1. Maintains the same technical depth and topic
2. Shows productive progression (not circular)
3. Follows the established interaction style
4. Does NOT introduce unnecessary questions from the assistant

Generate both the next user message and assistant response.

Output format:
USER: [next user message]
ASSISTANT: [assistant response that executes without asking permission]
"""

async def generate_extension(
    self,
    path: ConversationPath,
    max_turns: int = 2
) -> list[dict]:
    """Generate extension turns for a path."""

    # Build context from path
    context = [
        {"role": node.role, "content": node.content}
        for node in path.nodes[-6:]
    ]

    context_str = "\n\n".join([
        f"{'User' if m['role'] == 'user' else 'Assistant'}: {m['content'][:400]}"
        for m in context
    ])

    extensions = []

    for _ in range(max_turns):
        response = await self.openai.chat.completions.create(
            model="gpt-5.2",
            messages=[
                {"role": "system", "content": EXTENSION_SYSTEM_PROMPT},
                {"role": "user", "content": f"""CONVERSATION:
{context_str}

Generate the next exchange:"""}
            ],
            temperature=0.5,
        )

        # Parse response
        content = response.choices[0].message.content

        user_match = re.search(r'USER:\s*(.+?)(?=ASSISTANT:|$)', content, re.DOTALL)
        assistant_match = re.search(r'ASSISTANT:\s*(.+)', content, re.DOTALL)

        if user_match and assistant_match:
            user_content = user_match.group(1).strip()
            assistant_content = assistant_match.group(1).strip()
```

```
            extensions.append({"role": "user", "content": user_content})
            extensions.append({"role": "assistant", "content": assistant_content})

            # Update context for next iteration
            context_str += f"\n\nUser: {user_content}\n\nAssistant: {assistant_content}"
        else:
            break

    return extensions
```

3.3.3. Creating Extension Records

```
async def _generate_extensions(
    self,
    paths: list[ConversationPath]
) -> list[SyntheticBranch]:
    """Generate extensions for high-quality paths."""

    branches = []

    # Filter to high-quality paths only
    quality_paths = [p for p in paths if (p.quality_score or 0) >= 0.7]

    for path in quality_paths[:10]:  # Limit extensions
        extensions = await self.generate_extension(
            path,
            self.config.extension_max_turns
        )

        if not extensions:
            continue

        # Build complete message history
        original_messages = [
            {"role": node.role, "content": node.content}
            for node in path.nodes
        ]

        combined = original_messages + extensions

        # Compute coordinates for last turn
        last_node = path.nodes[-1]
        last_extension = extensions[-1]["content"]

        branch = SyntheticBranch(
            branch_type="extension",
            original_conversation_id=path.conversation_id,
            parent_node_id=last_node.turn_id,
            messages=combined,
            coordinates=self.compute_synthetic_coordinates(
                last_node, last_extension, "continuation"
            ),
            phase_id=min(5, self._get_phase(last_node) + 1),  # Advance phase
            question_policy="no_questions",
            directive_completeness=0.8,  # High for extensions
            is_gold=True,
        )
        branches.append(branch)

    return branches
```

3.4. Trajectory-Contrast Pairs

3.4.1. Purpose

- Teach phase-appropriate behavior
- Same prompt, different response based on phase
- Model learns context sensitivity

3.4.2. Contrast Generation

```
async def generate_contrast_pair(
    self,
    prompt: str,
    phase_a: int,
    phase_b: int
) -> tuple[str, str]:
    """Generate contrasting responses for different phases."""

    response_a = await self._generate_phase_response(prompt, phase_a)
    response_b = await self._generate_phase_response(prompt, phase_b)

    return response_a, response_b

async def _generate_phase_response(
    self,
    prompt: str,
    phase_id: int
) -> str:
    """Generate response appropriate for a specific phase."""

    phase_descriptions = {
        0: "Opening phase - gathering context, clarifying questions acceptable",
        1: "Context phase - deep understanding, some clarification OK",
        2: "Solution phase - actively solving, NO questions, execute immediately",
        3: "Refinement phase - iterating on solution, NO questions, just improve",
        4: "Synthesis phase - summarizing, NO questions, produce deliverables",
        5: "Conclusion phase - final output, NO questions, complete the task",
    }

    question_policy = self.config.phase_question_policies.get(phase_id, "no_questions")

    system_prompt = f"""You are responding in the {phase_descriptions.get(phase_id, 'unknown')} of

Question policy: {question_policy}

{"You MAY ask clarifying questions if genuinely needed." if question_policy == "questions_if_requir

Respond to the user's message appropriately for this phase.
"""

    response = await self.openai.chat.completions.create(
        model="gpt-5.2",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": prompt}
        ],
        temperature=0.4,
    )

    return response.choices[0].message.content
```

---

4. Policy Enforcement

4.1. Question Policy by Phase

4.1.1. Policy Determination

```
def _get_question_policy(self, node: PathNode) -> str:
    """Determine question policy for a node based on phase."""

    phase_id = self._get_phase(node)
    return self.config.phase_question_policies.get(phase_id, "no_questions")
```

4.1.2. Policy Validation

```
def validate_question_policy(
    self,
    response: str,
    policy: str
) -> tuple[bool, list[str]]:
    """Validate response against question policy."""

    from .corpus_surgery.classifier import (
        compute_stall_score,
        ends_with_question,
        STRONG_PERMISSION_PHRASES,
    )

    errors = []

    if policy == "no_questions":
        # Must not ask any questions
        if ends_with_question(response):
            errors.append("Response ends with question (policy: no_questions)")

        stall_score = compute_stall_score(response)
        if stall_score >= 2:
            errors.append(f"Response has high stall score: {stall_score}")

        # Check for permission phrases
        response_lower = response.lower()
        for phrase in STRONG_PERMISSION_PHRASES:
            if phrase in response_lower:
                errors.append(f"Contains permission phrase: '{phrase}'")
                break

    elif policy == "questions_if_required":
        # Questions allowed but should not be gratuitous
        stall_score = compute_stall_score(response)
        if stall_score >= 5:  # Higher threshold
            errors.append(f"Too many permission-seeking phrases: {stall_score}")

    # questions_allowed - no restrictions

    return len(errors) == 0, errors
```

4.2. Repair Elimination

4.2.1. Detecting Repair Turns

```
def is_repair_turn(self, user_message: str, preceding_assistant: str) -> bool:
    """Detect if user is repairing/correcting the assistant."""

    from .corpus_surgery.quarantine import FRUSTRATION_TRIGGERS

    user_lower = user_message.lower()

    # Check for frustration triggers
    for trigger in FRUSTRATION_TRIGGERS:
        if trigger in user_lower:
            return True

    # Check for correction patterns
    correction_patterns = [
        r"^no[,\.]",              # "No, I meant..."
        r"^actually",             # "Actually..."
        r"that's not",            # "That's not what I asked"
        r"i already",             # "I already told you..."
        r"try again",             # "Try again"
        r"let me rephrase",       # "Let me rephrase..."
    ]

    for pattern in correction_patterns:
        if re.search(pattern, user_lower):
            return True

    return False
```

4.2.2. Generating Non-Repair Trajectories

```
async def generate_non_repair_trajectory(
    self,
    conversation: list[dict],
    repair_idx: int
) -> list[dict]:
    """Generate trajectory where repair was never needed."""

    # Get context up to the bad assistant turn
    context = conversation[:repair_idx - 1]  # Exclude bad turn

    # Get the user message before the bad turn
    user_message = conversation[repair_idx - 2]["content"]

    # Generate ideal response
    ideal = await self.generate_ideal_response(
        PathNode(content=conversation[repair_idx - 1]["content"], role="assistant"),
        context,
        {}
    )

    # Continue conversation naturally
    new_trajectory = context + [
        {"role": "assistant", "content": ideal}
    ]

    # Generate follow-up
    extensions = await self.generate_extension_from_messages(
        new_trajectory,
        max_turns=1
    )

    return new_trajectory + extensions
```

4.3. Format Lock

4.3.1. Detecting Format Constraints

```
def extract_format_constraints(self, user_message: str) -> dict:
    """Extract format constraints from user message."""

    constraints = {
        "forbid_bullets": False,
        "require_numbered": False,
        "must_return_code": False,
        "must_return_diff": False,
        "must_return_json": False,
        "must_not_omit": False,
    }

    user_lower = user_message.lower()

    # Check each constraint
    if any(p in user_lower for p in ["no bullet", "don't use bullet", "without bullet"]):
        constraints["forbid_bullets"] = True

    if any(p in user_lower for p in ["numbered list", "numbered steps", "number them"]):
        constraints["require_numbered"] = True

    if any(p in user_lower for p in ["in code", "write code", "implement", "function"]):
        constraints["must_return_code"] = True

    if any(p in user_lower for p in ["as json", "in json", "json format"]):
        constraints["must_return_json"] = True

    if any(p in user_lower for p in ["don't omit", "don't skip", "include everything", "full", "com
        constraints["must_not_omit"] = True

    return constraints
```

4.3.2. Enforcing Format in Generation

```
def build_format_instruction(self, constraints: dict) -> str:
    """Build format instruction for generation."""

    instructions = []

    if constraints.get("forbid_bullets"):
        instructions.append("Do NOT use bullet points. Use prose or numbered lists instead.")

    if constraints.get("require_numbered"):
        instructions.append("Use numbered lists for any structured content.")

    if constraints.get("must_return_code"):
        instructions.append("Include code in your response.")

    if constraints.get("must_return_json"):
        instructions.append("Return output in valid JSON format.")

    if constraints.get("must_not_omit"):
        instructions.append("Include ALL content - do not summarize or omit anything.")

    return "\n".join(instructions) if instructions else ""
```

---

5. Output Records

5.1. SFT Turn Records

```
def create_sft_record(
    self,
    branch: SyntheticBranch
) -> dict:
    """Create SFT record from synthetic branch."""

    return {
        "schema_version": "ctv3.1",
        "record_id": str(uuid4()),
        "record_type": "sft_turn",
        "source": {
            "origin": "convo_worm",
            "provider": "gpt-5.2",
            "source_id": branch.original_conversation_id,
            "created_at_utc": datetime.utcnow().isoformat(),
        },
        "context": {
            "domain": "mixed",
            "language": "en",
            "topology": {
                "coords_5d": branch.coordinates.to_list(),
                "phase_id": branch.phase_id,
                "homogeneity": branch.coordinates.z,
                "depth_norm": branch.coordinates.x / 10,  # Normalize
                "sibling_order": branch.coordinates.y,
                "temporal_norm": branch.coordinates.t,
                "complexity": branch.coordinates.n,
            },
            "policy": {
                "question_policy": branch.question_policy,
                "directive_completeness": branch.directive_completeness,
                "must_not_omit": False,
                "format_constraints": {},
            },
        },
        "input": {
            "messages": branch.messages[:-1],  # All but last
            "attachments": [],
        },
        "target": {
            "assistant_content": branch.messages[-1]["content"],
            "structured": {},
        },
        "tags": {
            "task_type": "respond",
            "prompt_class": "directive" if branch.directive_completeness >= 0.7 else "ambiguous",
            "branch_type": branch.branch_type,
        },
        "quality": {
            "gold": branch.is_gold,
            "weight": 1.0 if branch.is_gold else 0.5,
            "review_status": "auto",
            "failure_modes": [],
        },
    }
```

5.2. DPO Pair Records

```
def create_dpo_record(
    self,
    original_messages: list[dict],
    preferred_response: str,
    dispreferred_response: str,
    branch: SyntheticBranch
) -> dict:
    """Create DPO pair record."""

    return {
        "schema_version": "ctv3.1",
        "record_id": str(uuid4()),
        "record_type": "dpo_pair",
        "source": {
            "origin": "convo_worm",
            "provider": "gpt-5.2",
            "source_id": branch.original_conversation_id,
            "created_at_utc": datetime.utcnow().isoformat(),
        },
        "context": {
            "domain": "mixed",
            "language": "en",
            "topology": {
                "coords_5d": branch.coordinates.to_list(),
                "phase_id": branch.phase_id,
            },
            "policy": {
                "question_policy": "no_questions",
                "directive_completeness": branch.directive_completeness,
            },
        },
        "input": {
            "messages": original_messages,
            "attachments": [],
        },
        "candidates": {
            "preferred": {"assistant_content": preferred_response},
            "dispreferred": {"assistant_content": dispreferred_response},
        },
        "tags": {
            "task_type": "respond",
            "prompt_class": "directive",
            "dpo_reason": "friction_repair",
        },
        "quality": {
            "gold": True,
            "weight": 1.0,
            "review_status": "auto",
            "failure_modes": [],
        },
    }
```

6. Complete Pipeline

```
class ConversationWormPipeline:
    """Complete Conversation Worm pipeline."""

    async def run(
        self,
        conversation_ids: list[str] | None = None,
        output_dir: Path = None,
    ) -> dict:
        """Run the complete pipeline."""

        worm = ConversationWorm(self.supabase_client)

        # Get conversations to process
        if conversation_ids is None:
            conversation_ids = await self._get_all_conversation_ids()

        sft_records = []
        dpo_records = []

        for conv_id in conversation_ids:
            try:
                branches = await worm.process_conversation(conv_id)

                for branch in branches:
                    # Create SFT record
                    sft = worm.create_sft_record(branch)
                    sft_records.append(sft)

                    # Create DPO pair if it's an ideal response
                    if branch.branch_type == "ideal_response":
                        # Get original dispreferred response
                        original = self._get_original_response(conv_id, branch.parent_node_id)

                        if original:
                            dpo = worm.create_dpo_record(
                                branch.messages[:-1],
                                branch.messages[-1]["content"],  # Preferred
                                original,  # Dispreferred
                                branch,
                            )
                            dpo_records.append(dpo)

            except Exception as e:
                logger.warning(f"Error processing {conv_id}: {e}")
                continue

        # Export
        if output_dir:
            self._export_jsonl(sft_records, output_dir / "convo_sft.jsonl")
            self._export_jsonl(dpo_records, output_dir / "convo_dpo.jsonl")

        return {
            "conversations_processed": len(conversation_ids),
            "sft_records": len(sft_records),
            "dpo_pairs": len(dpo_records),
            "branches_by_type": self._count_by_type(worm.generated_branches),
        }
```

Phase 2C: Enhancer Agent

> **Purpose**: Canonicalize messy assistant outputs into clean training targets, complete unfinished code/plans, and create evaluation-grade hard prompts from historical failures.
>
> **Model**: GPT 5.2 (general augmentation)
>
> **Implementation File**: `rag_plusplus/ml/cognitivetwin_v3/worms/enhancer_agent.py`

---

1. Purpose

1.1. Canonicalize Outputs (Reduce Entropy)

1.1.1. Problem: Mixed Styles from Multiple Providers

- Training data contains outputs from ChatGPT, Claude, OpenAI API
- Each provider has different quirks and habits
- Inconsistent style reduces model coherence
- Provider-isms contaminate the learned behavior

1.1.2. Canonicalization Goals

- Remove provider-specific phrases
- Standardize opening/closing patterns
- Enforce consistent formatting
- Reduce permission-seeking language
- Maintain semantic content

1.1.3. Style Tether

- All outputs normalized to target style
- No "As an AI language model..."
- No excessive apologies
- No unnecessary hedging
- Direct, professional tone

1.2. Complete Unfinished Content

1.2.1. Types of Incomplete Content

- Partial code implementations
- Undetermined paths ("we'll do this later")
- Placeholder sections
- Incomplete plans/specs
- Truncated outputs

1.2.2. Completion Strategy

- Infer intent from context
- Apply assumption protocol (state, then execute)
- Match existing patterns and style
- Produce complete, runnable artifacts

1.2.3. Quality Requirements

- Completed content must compile
- Must integrate with existing code
- Must not contradict prior decisions
- Must be consistent with project conventions

1.3. Create Evaluation-Grade Hard Prompts

1.3.1. Source: Historical Annoyances

- Prompts that triggered permission-seeking
- "Don't omit" prompts that got summaries
- Format constraints that were violated
- Directive prompts with option-dumping responses

1.3.2. Conversion to Eval Cases

- Extract prompt and context
- Define expected behaviors
- Define disallowed behaviors
- Optionally provide reference answer

1.3.3. Use Cases

- Regression testing new checkpoints

- DPO preference pair generation
- Policy scorer training data

---

2. Canonicalization Rules

2.1. Remove Provider-isms

2.1.1. Phrases to Remove

```
PROVIDER_ISMS = [
    # AI self-reference
    r"as an ai(?: language model)?",
    r"as a large language model",
    r"i'?m (?:just )?an ai",
    r"i don'?t have (?:personal )?(?:opinions|feelings|preferences)",
    r"i'?m not able to",
    r"i can'?t (?:actually )?(?:browse|access|see)",

    # Over-apologizing
    r"i apologize(?:,? but)?",
    r"i'?m sorry(?:,? but)?",
    r"sorry for (?:any )?confusion",
    r"my apologies",

    # Over-hedging
    r"it'?s worth noting that",
    r"it'?s important to (?:note|mention|remember) that",
    r"please (?:note|keep in mind) that",
    r"i should mention that",

    # Filler acknowledgments
    r"^(?:sure|certainly|absolutely|of course)[,!]?\s*",
    r"^great question[,!]?\s*",
    r"^that'?s a (?:great|good|interesting) (?:question|point)[,!]?\s*",

    # Capability disclaimers (when irrelevant)
    r"i can'?t browse the (?:web|internet)",
    r"i don'?t have access to (?:the internet|real-?time)",
    r"as of my (?:knowledge )?cutoff",
    r"my training data (?:only )?(?:goes|extends) up to",

    # Permission closers
    r"would you like me to[^?]*\??\s*$",
    r"do you want me to[^?]*\??\s*$",
    r"shall i[^?]*\??\s*$",
    r"should i[^?]*\??\s*$",
    r"let me know if you[^.]*\.\s*$",
    r"feel free to ask[^.]*\.\s*$",
]

def remove_provider_isms(text: str) -> str:
    """Remove provider-specific phrases from text."""

    result = text

    for pattern in PROVIDER_ISMS:
        result = re.sub(pattern, "", result, flags=re.IGNORECASE | re.MULTILINE)

    # Clean up multiple newlines
    result = re.sub(r'\n{3,}', '\n\n', result)

    # Clean up leading/trailing whitespace
    result = result.strip()

    return result
```

2.1.2. Over-Apologizing Patterns

```
APOLOGY_PATTERNS = [
    r"i apologize for (?:any )?(?:confusion|inconvenience|misunderstanding)",
    r"sorry (?:for|about) (?:the )?(?:confusion|delay|misunderstanding)",
    r"my (?:sincere )?apologies",
    r"i'?m (?:truly |very )?sorry",
    r"please accept my apologies",
]

def reduce_apologies(text: str) -> str:
    """Remove excessive apologies while preserving genuine ones."""

    # Count apologies
    apology_count = sum(
        len(re.findall(pattern, text, re.IGNORECASE))
        for pattern in APOLOGY_PATTERNS
    )

    if apology_count <= 1:
        return text

    # Remove all but first apology
    result = text
    for pattern in APOLOGY_PATTERNS:
        matches = list(re.finditer(pattern, result, re.IGNORECASE))
        if len(matches) > 1:
            # Keep first, remove rest
            for match in matches[1:]:
                result = result[:match.start()] + result[match.end():]

    return result
```

2.1.3. Over-Disclaiming Patterns

```
DISCLAIMER_PATTERNS = [
    r"please note that this is not (?:legal|medical|financial) advice",
    r"this should not be taken as (?:professional )?advice",
    r"consult (?:a|with a) (?:professional|expert|specialist)",
    r"i'?m not a (?:lawyer|doctor|financial advisor)",
    r"this is for (?:informational|educational) purposes only",
]

def remove_irrelevant_disclaimers(text: str, context: str) -> str:
    """Remove disclaimers that aren't relevant to the context."""

    # Check if context involves sensitive topics
    sensitive_topics = ["legal", "medical", "financial", "health", "investment"]
    is_sensitive = any(topic in context.lower() for topic in sensitive_topics)

    if is_sensitive:
        return text  # Keep disclaimers for sensitive content

    # Remove disclaimers
    result = text
    for pattern in DISCLAIMER_PATTERNS:
        result = re.sub(pattern, "", result, flags=re.IGNORECASE)

    return result.strip()
```

2.2. Standardize Opening Patterns

2.2.1. Filler Acknowledgments to Remove

```
FILLER_OPENINGS = [
    r"^sure[,!]?\s*",
    r"^certainly[,!]?\s*",
    r"^absolutely[,!]?\s*",
    r"^of course[,!]?\s*",
    r"^great[,!]?\s*",
    r"^alright[,!]?\s*",
    r"^okay[,!]?\s*",
    r"^yes[,!]?\s*",
    r"^i'?d be happy to\s*",
    r"^i'?ll be glad to\s*",
    r"^happy to help[,!]?\s*",
]

def remove_filler_openings(text: str) -> str:
    """Remove filler acknowledgment phrases at start."""

    result = text

    # Apply patterns in sequence (some may chain)
    for _ in range(3):  # Max 3 iterations
        old_result = result
        for pattern in FILLER_OPENINGS:
            result = re.sub(pattern, "", result, flags=re.IGNORECASE)

        if result == old_result:
            break

    return result.strip()
```

2.2.2. Preferred Opening Patterns

```
PREFERRED_OPENINGS = {
    # For implementations
    "implementation": "Here is the implementation:",
    "code": "```",  # Jump straight to code

    # For explanations
    "explanation": "",  # Start directly with content

    # For analyses
    "analysis": "Analysis:",

    # For summaries
    "summary": "",  # Start directly

    # For lists
    "list": "1.",  # Start with first item
}

def apply_preferred_opening(text: str, task_type: str) -> str:
    """Apply preferred opening pattern based on task type."""

    # Remove existing filler
    text = remove_filler_openings(text)

    preferred = PREFERRED_OPENINGS.get(task_type, "")

    if preferred and not text.startswith(preferred):
        # Check if opening is appropriate
        if task_type == "code" and "```" not in text[:100]:
            return text  # Don't add if no code
        elif preferred:
            return preferred + " " + text

    return text
```

2.3. Standardize Closing Patterns

2.3.1. Permission Closers to Remove

```
PERMISSION_CLOSERS = [
    r"let me know if you(?:'d like| want| need)[^.!]*[.!]?\s*$",
    r"feel free to (?:ask|reach out|let me know)[^.!]*[.!]?\s*$",
    r"(?:please )?don'?t hesitate to[^.!]*[.!]?\s*$",
    r"if you (?:have any|need)[^.!]*questions[^.!]*[.!]?\s*$",
    r"hope (?:this|that) helps[.!]?\s*$",
    r"i hope this (?:helps|answers)[^.!]*[.!]?\s*$",
    r"is there anything else[^?]*\??\s*$",
    r"would you like (?:me to|more)[^?]*\??\s*$",
]

def remove_permission_closers(text: str) -> str:
    """Remove permission-seeking closers."""

    result = text

    for pattern in PERMISSION_CLOSERS:
        result = re.sub(pattern, "", result, flags=re.IGNORECASE | re.MULTILINE)

    return result.rstrip()
```

2.3.2. Preferred Closing Patterns

```
def get_preferred_closing(task_type: str, has_code: bool) -> str | None:
    """Get preferred closing pattern."""

    if task_type == "implementation" and has_code:
        return None  # End with code block, no closer needed

    if task_type == "explanation":
        return None  # End with content

    if task_type == "analysis":
        return None  # End with findings

    return None  # Default: no forced closer
```

2.4. Enforce Consistent Formatting

2.4.1. Code Block Formatting

```
def standardize_code_blocks(text: str) -> str:
    """Standardize code block formatting."""

    # Ensure language specifier
    # Replace ``` with ```python for Python-looking code
    def add_language(match):
        content = match.group(1)

        # Detect language
        if re.search(r'def |import |class |print\(', content):
            return f"```python\n{content}\n```"
        elif re.search(r'function |const |let |var |=>', content):
            return f"```javascript\n{content}\n```"
        elif re.search(r'fn |let mut |impl |pub ', content):
            return f"```rust\n{content}\n```"
        else:
            return f"```\n{content}\n```"

    # Find bare code blocks
    text = re.sub(r'```\n([^`]+)\n```', add_language, text)

    return text
```

2.4.2. List Formatting

```
def standardize_lists(text: str, prefer_numbered: bool = False) -> str:
    """Standardize list formatting."""

    if prefer_numbered:
        # Convert bullet lists to numbered
        lines = text.split('\n')
        result_lines = []
        counter = 0

        for line in lines:
            if re.match(r'^\s*[-*•]\s+', line):
                counter += 1
                line = re.sub(r'^\s*[-*•]\s+', f'{counter}. ', line)
            elif not line.strip():
                counter = 0  # Reset on empty line
            result_lines.append(line)

        return '\n'.join(result_lines)

    return text
```

2.4.3. Whitespace Normalization

```
def normalize_whitespace(text: str) -> str:
    """Normalize whitespace."""

    # Collapse multiple blank lines to 2
    text = re.sub(r'\n{3,}', '\n\n', text)

    # Remove trailing whitespace on lines
    lines = [line.rstrip() for line in text.split('\n')]
    text = '\n'.join(lines)

    # Ensure single newline at end
    text = text.rstrip() + '\n'

    return text
```

---

3. Unfinished Code Completion

3.1. Detecting Unfinished Content

3.1.1. Incomplete Code Markers

```
INCOMPLETE_CODE_MARKERS = [
    r'#\s*TODO:?\s*',
    r'#\s*FIXME:?\s*',
    r'#\s*XXX:?\s*',
    r'#\s*\.\.\.\s*$',
    r'pass\s*#\s*(?:implement|todo)',
    r'raise NotImplementedError',
    r'\.\.\.  # ',
]

def find_incomplete_code(text: str) -> list[dict]:
    """Find incomplete code sections."""

    incompletes = []

    # Extract code blocks
    code_blocks = re.findall(r'```(?:\w*)\n([\s\S]*?)\n```', text)

    for i, block in enumerate(code_blocks):
        for pattern in INCOMPLETE_CODE_MARKERS:
            matches = re.finditer(pattern, block, re.IGNORECASE)
            for match in matches:
                incompletes.append({
                    "block_index": i,
                    "marker": match.group(),
                    "position": match.start(),
                    "context": block[max(0, match.start()-50):match.end()+50],
                })

    return incompletes
```

3.1.2. Placeholder Patterns

```
PLACEHOLDER_PATTERNS = [
    r'\[TODO:?\s*[^\]]+\]',
    r'\[INSERT\s+[^\]]+\]',
    r'\[PLACEHOLDER\]',
    r'<\s*YOUR\s+[^>]+>',
    r'\.\.\.',  # Ellipsis (context-dependent)
]

def find_placeholders(text: str) -> list[dict]:
    """Find placeholder sections in text."""

    placeholders = []

    for pattern in PLACEHOLDER_PATTERNS:
        matches = re.finditer(pattern, text, re.IGNORECASE)
        for match in matches:
            placeholders.append({
                "placeholder": match.group(),
                "position": match.start(),
                "context": text[max(0, match.start()-100):match.end()+100],
            })

    return placeholders
```

3.1.3. Undetermined Path Indicators

```
UNDETERMINED_PATH_PATTERNS = [
    r"we(?:'ll| will) (?:do|handle|address) (?:this|that) later",
    r"this (?:needs to|should) be (?:implemented|completed)",
    r"(?:more|further) work (?:is )?needed",
    r"to be (?:determined|decided)",
    r"TBD",
    r"(?:left|leaving) (?:this|that) for (?:later|now)",
]

def find_undetermined_paths(text: str) -> list[dict]:
    """Find undetermined path indicators."""

    paths = []

    for pattern in UNDETERMINED_PATH_PATTERNS:
        matches = re.finditer(pattern, text, re.IGNORECASE)
        for match in matches:
            paths.append({
                "indicator": match.group(),
                "position": match.start(),
                "context": text[max(0, match.start()-150):match.end()+150],
            })

    return paths
```

3.2. Completion Generation

3.2.1. Code Completion System Prompt

```
CODE_COMPLETION_SYSTEM_PROMPT = """You are completing unfinished code for CognitiveTwin V3 training

RULES:
1. Complete the code to make it functional and runnable
2. Follow the existing code style and patterns
3. Use only dependencies that are already imported or commonly available
4. State any assumptions briefly in a comment
5. Do NOT ask questions - just complete the code

Output the completed code block only, ready to replace the original.
"""
```

3.2.2. Completing Code Blocks

```
async def complete_code_block(
    self,
    incomplete_code: str,
    context: str,
    marker: str
) -> str:
    """Complete an incomplete code block."""

    response = await self.openai.responses.create(
        model="gpt-5.2-codex",
        input=f"""CONTEXT:
{context}

INCOMPLETE CODE (contains {marker}):
```

{incomplete_code}

```
Complete this code. Replace all TODO/placeholder markers with working implementations.
Output only the completed code:""",
        temperature=0.2,
    )

    return response.output
```

3.2.3. Completing Prose Sections

```
PROSE_COMPLETION_PROMPT = """You are completing unfinished prose content for CognitiveTwin V3.

RULES:
1. Complete the section coherently
2. Match the existing style and tone
3. Be specific and detailed
4. Do NOT add permission-seeking or hedging language
5. Do NOT ask questions

Output only the completed section.
"""

async def complete_prose_section(
    self,
    incomplete_text: str,
    placeholder: str,
    context: str
) -> str:
    """Complete an incomplete prose section."""

    response = await self.openai.chat.completions.create(
        model="gpt-5.2",
        messages=[
            {"role": "system", "content": PROSE_COMPLETION_PROMPT},
            {"role": "user", "content": f"""CONTEXT:
{context}

INCOMPLETE TEXT (contains placeholder "{placeholder}"):
{incomplete_text}

Complete this text. Replace the placeholder with appropriate content.
Output only the completed text:"""}
        ],
        temperature=0.3,
    )

    return response.choices[0].message.content
```

3.3. Validation

3.3.1. Code Compilation Check

```
def validate_completed_code(self, code: str, language: str = "python") -> tuple[bool, str]:
    """Validate that completed code compiles."""

    if language == "python":
        try:
            import ast
            ast.parse(code)
            return True, ""
        except SyntaxError as e:
            return False, f"Syntax error: {e}"

    # Add other language validators as needed
    return True, ""  # Assume valid for unknown languages
```

3.3.2. Completeness Check

```
def validate_completeness(self, text: str) -> tuple[bool, list[str]]:
    """Validate that all incomplete markers have been resolved."""

    remaining = []

    # Check for remaining incomplete markers
    incomplete_code = find_incomplete_code(text)
    if incomplete_code:
        remaining.extend([m["marker"] for m in incomplete_code])

    placeholders = find_placeholders(text)
    if placeholders:
        remaining.extend([p["placeholder"] for p in placeholders])

    undetermined = find_undetermined_paths(text)
    if undetermined:
        remaining.extend([u["indicator"] for u in undetermined])

    return len(remaining) == 0, remaining
```

---

4. Regression Test Extraction

4.1. Identifying Annoyance Cases

4.1.1. Permission-Seeking Annoyances

```
def find_permission_annoyances(
    self,
    conversations: list[dict]
) -> list[dict]:
    """Find cases where model asked permission inappropriately."""

    annoyances = []

    for conv in conversations:
        for i, turn in enumerate(conv["turns"]):
            if turn["role"] != "assistant":
                continue

            # Get preceding user message
            user_turn = conv["turns"][i-1] if i > 0 else None
            if not user_turn or user_turn["role"] != "user":
                continue

            # Classify
            from .corpus_surgery.classifier import classify_assistant_turn

            result = classify_assistant_turn(
                assistant_message=turn["content"],
                user_message=user_turn["content"],
                phase_id=turn.get("phase_id", 2),
                format_constraints={},
                directive_completeness=self._compute_completeness(user_turn["content"]),
            )

            if result.classification.value == "unjustified":
                annoyances.append({
                    "type": "permission_seeking",
                    "conversation_id": conv["id"],
                    "turn_index": i,
                    "user_message": user_turn["content"],
                    "assistant_message": turn["content"],
                    "classification": result,
                })

    return annoyances
```

4.1.2. Omission Annoyances

```
def find_omission_annoyances(
    self,
    conversations: list[dict]
) -> list[dict]:
    """Find cases where model omitted content despite 'don't omit' instruction."""

    annoyances = []

    for conv in conversations:
        for i, turn in enumerate(conv["turns"]):
            if turn["role"] != "user":
                continue

            # Check for "don't omit" instructions
            if not self._has_dont_omit(turn["content"]):
                continue

            # Get following assistant response
            if i + 1 >= len(conv["turns"]):
                continue

            assistant_turn = conv["turns"][i + 1]
            if assistant_turn["role"] != "assistant":
                continue

            # Check for omission indicators
            if self._has_omission_indicators(assistant_turn["content"]):
                annoyances.append({
                    "type": "omission",
                    "conversation_id": conv["id"],
                    "turn_index": i + 1,
                    "user_message": turn["content"],
                    "assistant_message": assistant_turn["content"],
                    "omission_indicators": self._get_omission_indicators(assistant_turn["content"])
                })

    return annoyances

def _has_dont_omit(self, text: str) -> bool:
    """Check if text contains 'don't omit' instruction."""
    patterns = [
        r"don'?t omit",
        r"don'?t skip",
        r"include (?:everything|all)",
        r"full (?:content|text|code)",
        r"complete (?:content|text|code)",
        r"no summariz",
        r"exact (?:copy|rewrite)",
    ]
    text_lower = text.lower()
    return any(re.search(p, text_lower) for p in patterns)

def _has_omission_indicators(self, text: str) -> bool:
    """Check if text indicates omission."""
    indicators = [
        r"\.\.\.",
        r"\[\.\.\.omitted\.\.\.\]",
        r"\[rest of",
```

```
        r"(?:and so on|etc\.)",
        r"similar(?:ly)?(?:,| to)",
        r"the (?:rest|remainder)",
        r"continue(?:s|d)? (?:similarly|as|with)",
    ]
    text_lower = text.lower()
    return any(re.search(p, text_lower) for p in indicators)
```

4.1.3. Format Drift Annoyances

```
def find_format_drift_annoyances(
    self,
    conversations: list[dict]
) -> list[dict]:
    """Find cases where model violated format constraints."""

    annoyances = []

    for conv in conversations:
        for i, turn in enumerate(conv["turns"]):
            if turn["role"] != "user":
                continue

            # Extract format constraints
            constraints = self._extract_format_constraints(turn["content"])

            if not any(constraints.values()):
                continue

            # Get following assistant response
            if i + 1 >= len(conv["turns"]):
                continue

            assistant_turn = conv["turns"][i + 1]
            if assistant_turn["role"] != "assistant":
                continue

            # Check for violations
            violations = self._check_format_violations(
                assistant_turn["content"],
                constraints
            )

            if violations:
                annoyances.append({
                    "type": "format_drift",
                    "conversation_id": conv["id"],
                    "turn_index": i + 1,
                    "user_message": turn["content"],
                    "assistant_message": assistant_turn["content"],
                    "constraints": constraints,
                    "violations": violations,
                })

    return annoyances
```

4.2. Creating Eval Cases

4.2.1. Eval Case Schema

```
@dataclass
class EvalCase:
    """Evaluation case for regression testing."""

    record_type: str = "eval_case"
    case_id: str = ""
    case_type: str = ""  # permission_seeking, omission, format_drift

    # Input
    prompt: str = ""
    context: list[dict] = field(default_factory=list)
    format_constraints: dict = field(default_factory=dict)

    # Expected behavior
    expected_behaviors: list[str] = field(default_factory=list)
    disallowed_behaviors: list[str] = field(default_factory=list)
    disallowed_phrases: list[str] = field(default_factory=list)

    # Validation rules
    must_not_end_with_question: bool = True
    must_contain_artifact: bool = False
    must_follow_format: str = ""

    # Reference answer (optional)
    reference_answer: str = ""

    # Source
    source_conversation: str = ""
    source_turn: int = 0
```

4.2.2. Generating Eval Cases from Annoyances

```
def create_eval_case_from_annoyance(
    self,
    annoyance: dict
) -> EvalCase:
    """Create eval case from an annoyance instance."""

    case = EvalCase(
        case_id=f"{annoyance['type']}_{annoyance['conversation_id']}_{annoyance['turn_index']}",
        case_type=annoyance["type"],
        prompt=annoyance["user_message"],
        source_conversation=annoyance["conversation_id"],
        source_turn=annoyance["turn_index"],
    )

    if annoyance["type"] == "permission_seeking":
        case.expected_behaviors = [
            "Execute immediately without asking permission",
            "Produce the requested output directly",
            "State assumptions as declarations if needed",
        ]
        case.disallowed_behaviors = [
            "Ask for confirmation before proceeding",
            "Offer multiple options without choosing",
            "End with a question",
        ]
        case.disallowed_phrases = [
            "would you like me to",
            "should i",
            "do you want me to",
            "before i proceed",
            "can you confirm",
        ]
        case.must_not_end_with_question = True

    elif annoyance["type"] == "omission":
        case.expected_behaviors = [
            "Include ALL content without summarizing",
            "Do not use ellipsis (...) to skip content",
            "Preserve complete text/code",
        ]
        case.disallowed_behaviors = [
            "Summarize when full content requested",
            "Use [...] or ... to skip sections",
            "Say 'and so on' or 'etc.'",
        ]
        case.disallowed_phrases = [
            "...",
            "[...]",
            "and so on",
            "etc.",
            "rest of the",
            "continues similarly",
        ]

    elif annoyance["type"] == "format_drift":
        constraints = annoyance.get("constraints", {})

        case.format_constraints = constraints
```

```
        case.expected_behaviors = []
        case.disallowed_behaviors = []

        if constraints.get("forbid_bullets"):
            case.expected_behaviors.append("Use numbered lists or prose instead of bullets")
            case.disallowed_behaviors.append("Use bullet points")

        if constraints.get("require_numbered"):
            case.expected_behaviors.append("Use numbered lists for structured content")
            case.disallowed_behaviors.append("Use bullet points or unstructured prose")

        if constraints.get("must_return_json"):
            case.expected_behaviors.append("Return valid JSON")
            case.disallowed_behaviors.append("Return non-JSON formatted output")
            case.must_follow_format = "json"

    return case
```

4.3. Generating Reference Answers

4.3.1. Reference Answer Generation

```
REFERENCE_ANSWER_PROMPT = """You are generating a reference answer for a CognitiveTwin V3 evaluatio

The original assistant response violated the expected behavior. Generate the CORRECT response.

RULES:
{rules}

Generate the correct response that satisfies all requirements:
"""

async def generate_reference_answer(
    self,
    eval_case: EvalCase
) -> str:
    """Generate reference answer for eval case."""

    rules = []
    for behavior in eval_case.expected_behaviors:
        rules.append(f"- MUST: {behavior}")
    for behavior in eval_case.disallowed_behaviors:
        rules.append(f"- MUST NOT: {behavior}")
    for phrase in eval_case.disallowed_phrases:
        rules.append(f"- FORBIDDEN PHRASE: '{phrase}'")

    if eval_case.must_not_end_with_question:
        rules.append("- MUST NOT end with a question mark")

    if eval_case.must_follow_format:
        rules.append(f"- MUST output in {eval_case.must_follow_format} format")

    context_str = ""
    if eval_case.context:
        context_str = "\n".join([
            f"{'User' if m['role'] == 'user' else 'Assistant'}: {m['content'][:300]}"
            for m in eval_case.context[-4:]
        ])

    response = await self.openai.chat.completions.create(
        model="gpt-5.2",
        messages=[
            {"role": "system", "content": REFERENCE_ANSWER_PROMPT.format(rules='\n'.join(rules))},
            {"role": "user", "content": f"""CONTEXT:
{context_str}

USER MESSAGE:
{eval_case.prompt}

FORMAT CONSTRAINTS: {eval_case.format_constraints}

Generate the correct assistant response:"""}
        ],
        temperature=0.3,
    )

    return response.choices[0].message.content
```

5. Complete Pipeline

```
class EnhancerAgentPipeline:
    """Complete Enhancer Agent pipeline."""

    def __init__(
        self,
        supabase_client,
        openai_client: OpenAI = None,
    ):
        self.client = supabase_client
        self.openai = openai_client or OpenAI()
        self.enhancer = EnhancerAgent(openai_client=self.openai)

    async def run(
        self,
        conversations: list[dict],
        output_dir: Path,
    ) -> dict:
        """Run the complete enhancer pipeline."""

        enhanced_records = []
        eval_cases = []
        dpo_pairs = []

        # Phase 1: Canonicalize all assistant turns
        for conv in conversations:
            for turn in conv["turns"]:
                if turn["role"] == "assistant":
                    original = turn["content"]
                    canonicalized = self.enhancer.canonicalize(original, conv)

                    if canonicalized != original:
                        enhanced_records.append({
                            "original": original,
                            "enhanced": canonicalized,
                            "conversation_id": conv["id"],
                        })
                        turn["content"] = canonicalized

        # Phase 2: Complete unfinished content
        for conv in conversations:
            for turn in conv["turns"]:
                if turn["role"] == "assistant":
                    incompletes = find_incomplete_code(turn["content"])
                    incompletes.extend(find_placeholders(turn["content"]))

                    if incompletes:
                        completed = await self.enhancer.complete_all(
                            turn["content"],
                            self._get_context(conv, turn)
                        )

                        enhanced_records.append({
                            "original": turn["content"],
                            "enhanced": completed,
                            "type": "completion",
                        })
                        turn["content"] = completed
```

```
        # Phase 3: Extract regression test cases
        annoyances = []
        annoyances.extend(self.enhancer.find_permission_annoyances(conversations))
        annoyances.extend(self.enhancer.find_omission_annoyances(conversations))
        annoyances.extend(self.enhancer.find_format_drift_annoyances(conversations))

        for annoyance in annoyances:
            # Create eval case
            eval_case = self.enhancer.create_eval_case_from_annoyance(annoyance)

            # Generate reference answer
            eval_case.reference_answer = await self.enhancer.generate_reference_answer(eval_case)

            eval_cases.append(eval_case)

            # Create DPO pair
            dpo_pair = {
                "prompt": annoyance["user_message"],
                "preferred": eval_case.reference_answer,
                "dispreferred": annoyance["assistant_message"],
                "source": "enhancer_agent",
            }
            dpo_pairs.append(dpo_pair)

        # Export
        self._export_enhanced(enhanced_records, output_dir / "enhanced.jsonl")
        self._export_eval_cases(eval_cases, output_dir / "eval_regression.jsonl")
        self._export_dpo_pairs(dpo_pairs, output_dir / "enhancer_dpo.jsonl")

        return {
            "enhanced_records": len(enhanced_records),
            "eval_cases": len(eval_cases),
            "dpo_pairs": len(dpo_pairs),
            "annoyances_by_type": self._count_by_type(annoyances),
        }
```

---

6. Output Records

6.1. Enhanced SFT Records

```
def create_enhanced_sft_record(
    self,
    original: str,
    enhanced: str,
    conversation_id: str,
    context: list[dict]
) -> dict:
    """Create SFT record from enhanced content."""

    return {
        "schema_version": "ctv3.1",
        "record_id": str(uuid4()),
        "record_type": "sft_turn",
        "source": {
            "origin": "enhancer_agent",
            "provider": "gpt-5.2",
            "source_id": conversation_id,
            "created_at_utc": datetime.utcnow().isoformat(),
        },
        "context": {
            "domain": "mixed",
            "policy": {
                "question_policy": "no_questions",
                "directive_completeness": 0.8,
            },
        },
        "input": {
            "messages": context,
        },
        "target": {
            "assistant_content": enhanced,
        },
        "quality": {
            "gold": True,
            "weight": 1.0,
            "review_status": "auto",
            "enhancement_type": "canonicalization",
        },
    }
```

6.2. Eval Case Records

```
def export_eval_case(self, case: EvalCase) -> dict:
    """Export eval case to CTv3.1 format."""

    return {
        "schema_version": "ctv3.1",
        "record_id": case.case_id,
        "record_type": "eval_case",
        "source": {
            "origin": "enhancer_agent",
            "source_id": case.source_conversation,
        },
        "input": {
            "messages": case.context + [{"role": "user", "content": case.prompt}],
        },
        "context": {
            "policy": {
                "question_policy": "no_questions",
            },
            "format_constraints": case.format_constraints,
        },
        "checks": {
            "expected_behaviors": case.expected_behaviors,
            "disallowed_behaviors": case.disallowed_behaviors,
            "disallowed_phrases": case.disallowed_phrases,
            "must_not_end_with_question": case.must_not_end_with_question,
            "must_follow_format": case.must_follow_format,
        },
        "reference": {
            "answer": case.reference_answer,
        },
        "quality": {
            "gold": True,
            "weight": 0.0,  # Not trained on
            "review_status": "auto",
        },
    }
```

Phase 3: Dataset Builder

> **Purpose**: Define the CTv3.1 JSONL schema, implement policy labeling, generate DPO pairs for all failure modes, and export to train/dpo/eval splits.
>
> **Implementation Files**: - `rag_plusplus/ml/cognitivetwin_v3/schema.py` - `rag_plusplus/ml/cognitivetwin_v3/dataset/labeler.py` - `rag_plusplus/ml/cognitivetwin_v3/dataset/pair_generator.py` - `rag_plusplus/ml/cognitivetwin_v3/dataset/exporter.py`

---

1. CTv3.1 JSONL Schema

1.1. Schema Version and Record Types

```
from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum
from typing import Any, Optional
from uuid import uuid4

SCHEMA_VERSION = "ctv3.1"

class RecordType(str, Enum):
    SFT_TURN = "sft_turn"
    DPO_PAIR = "dpo_pair"
    EVAL_CASE = "eval_case"
    REPO_TASK = "repo_task"
```

1.2. Source Information

```
class SourceOrigin(str, Enum):
    HUMAN_CORPUS = "human_corpus"
    REPO_WORM = "repo_worm"
    CONVO_WORM = "convo_worm"
    ENHANCER_AGENT = "enhancer_agent"

class SourceProvider(str, Enum):
    CHATGPT = "chatgpt"
    CLAUDE = "claude"
    OPENAI = "openai"
    INTERNAL = "internal"

@dataclass
class SourceInfo:
    origin: SourceOrigin
    provider: SourceProvider
    source_id: str = ""
    created_at_utc: str = field(default_factory=lambda: datetime.utcnow().isoformat())

    def to_dict(self) -> dict:
        return {
            "origin": self.origin.value,
            "provider": self.provider.value,
            "source_id": self.source_id,
            "created_at_utc": self.created_at_utc,
        }
```

1.3. Context Information

1.3.1. Domain and Language

```
class Domain(str, Enum):
    CODE = "code"
    RESEARCH = "research"
    PLANNING = "planning"
    OPS = "ops"
    MIXED = "mixed"
```

1.3.2. Topology Coordinates

```
@dataclass
class TopologyCoords:
    """5D trajectory coordinates plus phase."""

    coords_5d: list[float] = field(default_factory=lambda: [0.0, 0.0, 0.5, 0.5, 1.0])
    phase_id: int = 2
    homogeneity: float = 0.5
    depth_norm: float = 0.0
    sibling_order: float = 0.0
    temporal_norm: float = 0.5
    complexity: float = 1.0

    def to_dict(self) -> dict:
        return {
            "coords_5d": self.coords_5d,
            "phase_id": self.phase_id,
            "homogeneity": self.homogeneity,
            "depth_norm": self.depth_norm,
            "sibling_order": self.sibling_order,
            "temporal_norm": self.temporal_norm,
            "complexity": self.complexity,
        }
```

1.3.3. Policy Information

```
class QuestionPolicy(str, Enum):
    NO_QUESTIONS = "no_questions"
    QUESTIONS_IF_REQUIRED = "questions_if_required"
    QUESTIONS_ALLOWED = "questions_allowed"

@dataclass
class FormatConstraints:
    forbid_bullets: bool = False
    require_numbered: bool = False
    must_return_code: bool = False
    must_return_diff: bool = False
    must_return_json: bool = False

    def to_dict(self) -> dict:
        return {
            "forbid_bullets": self.forbid_bullets,
            "require_numbered": self.require_numbered,
            "must_return_code": self.must_return_code,
            "must_return_diff": self.must_return_diff,
            "must_return_json": self.must_return_json,
        }

@dataclass
class PolicyInfo:
    question_policy: QuestionPolicy = QuestionPolicy.NO_QUESTIONS
    directive_completeness: float = 0.8
    must_not_omit: bool = False
    format_constraints: FormatConstraints = field(default_factory=FormatConstraints)

    def to_dict(self) -> dict:
        return {
            "question_policy": self.question_policy.value,
            "directive_completeness": self.directive_completeness,
            "must_not_omit": self.must_not_omit,
            "format_constraints": self.format_constraints.to_dict(),
        }

@dataclass
class ContextInfo:
    domain: Domain = Domain.MIXED
    language: str = "en"
    topology: TopologyCoords = field(default_factory=TopologyCoords)
    policy: PolicyInfo = field(default_factory=PolicyInfo)

    def to_dict(self) -> dict:
        return {
            "domain": self.domain.value,
            "language": self.language,
            "topology": self.topology.to_dict(),
            "policy": self.policy.to_dict(),
        }
```

1.4. Input Data

```
@dataclass
class Message:
    role: str  # "system", "user", "assistant"
    content: str

    def to_dict(self) -> dict:
        return {"role": self.role, "content": self.content}

@dataclass
class Attachment:
    type: str = "repo_context"
    repo: str = ""
    commit_sha: str = ""
    path: str = ""
    span: dict = field(default_factory=lambda: {"start_line": 0, "end_line": 0})
    content: str = ""

    def to_dict(self) -> dict:
        return {
            "type": self.type,
            "repo": self.repo,
            "commit_sha": self.commit_sha,
            "path": self.path,
            "span": self.span,
            "content": self.content,
        }

@dataclass
class InputData:
    messages: list[Message] = field(default_factory=list)
    attachments: list[Attachment] = field(default_factory=list)

    def to_dict(self) -> dict:
        return {
            "messages": [m.to_dict() for m in self.messages],
            "attachments": [a.to_dict() for a in self.attachments],
        }
```

1.5. Target Data

```
@dataclass
class StructuredOutput:
    diff_unified: str = ""
    json: dict = field(default_factory=dict)
    plan_steps: list[str] = field(default_factory=list)

    def to_dict(self) -> dict:
        return {
            "diff_unified": self.diff_unified,
            "json": self.json,
            "plan_steps": self.plan_steps,
        }

@dataclass
class TargetData:
    assistant_content: str = ""
    structured: StructuredOutput = field(default_factory=StructuredOutput)

    def to_dict(self) -> dict:
        return {
            "assistant_content": self.assistant_content,
            "structured": self.structured.to_dict(),
        }
```

1.6. Tags

```
class TaskType(str, Enum):
    REWRITE = "rewrite"
    IMPLEMENT = "implement"
    DEBUG = "debug"
    EXPLAIN = "explain"
    REFACTOR = "refactor"
    DESIGN = "design"
    EVALUATE = "evaluate"
    RESPOND = "respond"

class PromptClass(str, Enum):
    DIRECTIVE = "directive"
    AMBIGUOUS = "ambiguous"
    OPEN_ENDED = "open_ended"
    BLOCKED = "blocked"

@dataclass
class RepoTaskInfo:
    module: str = ""
    symbols: list[str] = field(default_factory=list)
    build_required: bool = False
    tests_required: bool = False

    def to_dict(self) -> dict:
        return {
            "module": self.module,
            "symbols": self.symbols,
            "build_required": self.build_required,
            "tests_required": self.tests_required,
        }

@dataclass
class TagInfo:
    task_type: TaskType = TaskType.RESPOND
    prompt_class: PromptClass = PromptClass.DIRECTIVE
    repo_task: RepoTaskInfo = field(default_factory=RepoTaskInfo)

    def to_dict(self) -> dict:
        return {
            "task_type": self.task_type.value,
            "prompt_class": self.prompt_class.value,
            "repo_task": self.repo_task.to_dict(),
        }
```

1.7. Quality

```
class ReviewStatus(str, Enum):
    AUTO = "auto"
    HUMAN_VERIFIED = "human_verified"
    REJECTED = "rejected"

class FailureMode(str, Enum):
    ASKED_PERMISSION = "asked_permission"
    ENDED_WITH_QUESTION = "ended_with_question"
    OMITTED_REQUIRED_CONTENT = "omitted_required_content"
    FORMAT_DRIFT = "format_drift"
    HALLUCINATED_REPO_FACTS = "hallucinated_repo_facts"

@dataclass
class QualityInfo:
    gold: bool = False
    weight: float = 1.0
    review_status: ReviewStatus = ReviewStatus.AUTO
    failure_modes: list[FailureMode] = field(default_factory=list)

    def to_dict(self) -> dict:
        return {
            "gold": self.gold,
            "weight": self.weight,
            "review_status": self.review_status.value,
            "failure_modes": [f.value for f in self.failure_modes],
        }
```

1.8. Complete CTv3 Record

```
@dataclass
class CTv3Record:
    """Complete CTv3.1 record."""

    schema_version: str = SCHEMA_VERSION
    record_id: str = field(default_factory=lambda: str(uuid4()))
    record_type: RecordType = RecordType.SFT_TURN

    source: SourceInfo = field(default_factory=SourceInfo)
    context: ContextInfo = field(default_factory=ContextInfo)
    input: InputData = field(default_factory=InputData)
    target: TargetData = field(default_factory=TargetData)
    tags: TagInfo = field(default_factory=TagInfo)
    quality: QualityInfo = field(default_factory=QualityInfo)

    def to_dict(self) -> dict:
        return {
            "schema_version": self.schema_version,
            "record_id": self.record_id,
            "record_type": self.record_type.value,
            "source": self.source.to_dict(),
            "context": self.context.to_dict(),
            "input": self.input.to_dict(),
            "target": self.target.to_dict(),
            "tags": self.tags.to_dict(),
            "quality": self.quality.to_dict(),
        }

    def to_json(self) -> str:
        import json
        return json.dumps(self.to_dict())
```

1.9. DPO Pair Record

```
@dataclass
class DPOCandidates:
    preferred: TargetData = field(default_factory=TargetData)
    dispreferred: TargetData = field(default_factory=TargetData)

    def to_dict(self) -> dict:
        return {
            "preferred": self.preferred.to_dict(),
            "dispreferred": self.dispreferred.to_dict(),
        }

@dataclass
class CTv3DPORecord:
    """DPO pair record."""

    schema_version: str = SCHEMA_VERSION
    record_id: str = field(default_factory=lambda: str(uuid4()))
    record_type: RecordType = RecordType.DPO_PAIR

    source: SourceInfo = field(default_factory=SourceInfo)
    context: ContextInfo = field(default_factory=ContextInfo)
    input: InputData = field(default_factory=InputData)
    candidates: DPOCandidates = field(default_factory=DPOCandidates)
    tags: TagInfo = field(default_factory=TagInfo)
    quality: QualityInfo = field(default_factory=QualityInfo)

    def to_dict(self) -> dict:
        return {
            "schema_version": self.schema_version,
            "record_id": self.record_id,
            "record_type": self.record_type.value,
            "source": self.source.to_dict(),
            "context": self.context.to_dict(),
            "input": self.input.to_dict(),
            "candidates": self.candidates.to_dict(),
            "tags": self.tags.to_dict(),
            "quality": self.quality.to_dict(),
        }
```

---

2. Policy Labeler

2.1. Directive Completeness Computation

```
import re

class DirectiveCompletenessLabeler:
    """Compute directive_completeness score for prompts."""

    # Imperative verbs that indicate clear directives
    IMPERATIVE_VERBS = [
        "rewrite", "generate", "implement", "create", "build",
        "write", "return", "extract", "convert", "transform",
        "refactor", "fix", "update", "add", "remove", "delete",
        "change", "modify", "replace", "debug", "test", "analyze",
        "explain", "summarize", "list", "show", "find", "search",
    ]

    # Format specification patterns
    FORMAT_PATTERNS = [
        r"in json",
        r"as json",
        r"return(?:ing)? json",
        r"as csv",
        r"in csv",
        r"as markdown",
        r"in markdown",
        r"don'?t omit",
        r"exact(?:ly)?",
        r"no bullet",
        r"numbered list",
        r"as code",
        r"in python",
        r"in typescript",
    ]

    # Missing input indicators
    MISSING_INPUT_PATTERNS = [
        r"(?:the |this |that )?code",  # References code not provided
        r"(?:the |this |that )?file",   # References file not provided
        r"(?:the |this |that )?function",
    ]

    def compute(self, user_message: str, context: dict = None) -> float:
        """Compute directive completeness score."""

        context = context or {}
        score = 0.0
        user_lower = user_message.lower()

        # +0.35 for imperative verb
        if self._has_imperative_verb(user_lower):
            score += 0.35

        # +0.25 for format specification
        if self._has_format_specification(user_lower):
            score += 0.25

        # +0.20 for complete inputs
        if self._has_required_inputs(user_message, context):
            score += 0.20
```

```
        # -0.40 for missing required inputs
        if self._missing_required_inputs(user_message, context):
            score -= 0.40

        # -0.20 for material ambiguity
        if self._has_material_ambiguity(user_lower):
            score -= 0.20

        return max(0.0, min(1.0, score))

    def _has_imperative_verb(self, text: str) -> bool:
        """Check for imperative verb at start or in command position."""

        for verb in self.IMPERATIVE_VERBS:
            # At start of sentence
            if re.search(rf'^{verb}\b', text):
                return True
            # After "please" or "can you"
            if re.search(rf'(?:please|can you)\s+{verb}\b', text):
                return True
            # After colon (in commands)
            if re.search(rf':\s*{verb}\b', text):
                return True

        return False

    def _has_format_specification(self, text: str) -> bool:
        """Check for format specification."""

        for pattern in self.FORMAT_PATTERNS:
            if re.search(pattern, text):
                return True

        return False

    def _has_required_inputs(self, text: str, context: dict) -> bool:
        """Check if required inputs are present."""

        # Check for code block in message
        has_code = bool(re.search(r"```[\s\S]*?```", text))

        # Check for file path
        has_file_path = bool(re.search(r"[/\\][\w./\\]+\.\w+", text))

        # Check for substantial text (> 200 chars)
        has_long_text = len(text) > 200

        # Check for attachments in context
        has_attachments = bool(context.get("attachments"))

        return has_code or has_file_path or has_long_text or has_attachments

    def _missing_required_inputs(self, text: str, context: dict) -> bool:
        """Check if required inputs are missing."""

        text_lower = text.lower()

        # Check for transformation words without input
        transformation_words = [
```

```
            "refactor", "rewrite", "transform", "convert",
            "enhance", "improve", "fix", "update"
        ]

        needs_input = any(word in text_lower for word in transformation_words)

        if needs_input and not self._has_required_inputs(text, context):
            # Check if it references something vague
            for pattern in self.MISSING_INPUT_PATTERNS:
                if re.search(pattern, text_lower):
                    return True

        return False

    def _has_material_ambiguity(self, text: str) -> bool:
        """Check for material ambiguity."""

        ambiguity_patterns = [
            r"this or that",
            r"either.+or",
            r"what (?:should|would)",
            r"which (?:one|approach|method)",
            r"how should i",
        ]

        return any(re.search(p, text) for p in ambiguity_patterns)
```

2.2. Question Policy Determination

```
class QuestionPolicyLabeler:
    """Determine question policy based on context."""

    # Phase -> default policy mapping
    PHASE_POLICIES = {
        0: QuestionPolicy.QUESTIONS_IF_REQUIRED,  # Opening
        1: QuestionPolicy.QUESTIONS_IF_REQUIRED,  # Context
        2: QuestionPolicy.NO_QUESTIONS,           # Solution
        3: QuestionPolicy.NO_QUESTIONS,           # Refinement
        4: QuestionPolicy.NO_QUESTIONS,           # Synthesis
        5: QuestionPolicy.NO_QUESTIONS,           # Conclusion
    }

    def compute(
        self,
        phase_id: int,
        directive_completeness: float,
        user_message: str
    ) -> QuestionPolicy:
        """Determine question policy."""

        # Check for explicit permission in user message
        if self._user_asked_for_options(user_message):
            return QuestionPolicy.QUESTIONS_ALLOWED

        # High directive completeness -> no questions
        if directive_completeness >= 0.7:
            return QuestionPolicy.NO_QUESTIONS

        # Low directive completeness -> questions if required
        if directive_completeness < 0.4:
            return QuestionPolicy.QUESTIONS_IF_REQUIRED

        # Medium completeness -> use phase default
        return self.PHASE_POLICIES.get(phase_id, QuestionPolicy.NO_QUESTIONS)

    def _user_asked_for_options(self, text: str) -> bool:
        """Check if user explicitly asked for options."""

        patterns = [
            r"what (?:are )?(?:my |the )?options",
            r"give me (?:some )?options",
            r"list (?:the |some )?options",
            r"what (?:could|can|should) i",
            r"what do you (?:think|suggest|recommend)",
        ]

        text_lower = text.lower()
        return any(re.search(p, text_lower) for p in patterns)
```

2.3. Format Constraints Extraction

```
class FormatConstraintsLabeler:
    """Extract format constraints from user message."""

    def extract(self, user_message: str) -> FormatConstraints:
        """Extract format constraints."""

        constraints = FormatConstraints()
        user_lower = user_message.lower()

        # Forbid bullets
        if any(p in user_lower for p in [
            "no bullet", "don't use bullet", "without bullet",
            "avoid bullet", "not bullet"
        ]):
            constraints.forbid_bullets = True

        # Require numbered
        if any(p in user_lower for p in [
            "numbered list", "numbered steps", "number them",
            "use numbers", "with numbers"
        ]):
            constraints.require_numbered = True

        # Must return code
        if any(p in user_lower for p in [
            "in code", "write code", "implement", "as code",
            "function", "class", "method"
        ]):
            constraints.must_return_code = True

        # Must return diff
        if any(p in user_lower for p in [
            "as diff", "in diff", "show diff", "unified diff"
        ]):
            constraints.must_return_diff = True

        # Must return JSON
        if any(p in user_lower for p in [
            "as json", "in json", "json format", "return json"
        ]):
            constraints.must_return_json = True

        return constraints
```

2.4. Complete Label Generator

```
@dataclass
class Labels:
    directive_completeness: float
    question_policy: QuestionPolicy
    format_constraints: FormatConstraints
    must_not_omit: bool
    prompt_class: PromptClass
    domain: Domain

class PolicyLabeler:
    """Complete policy labeler for CTv3 records."""

    def __init__(self):
        self.completeness_labeler = DirectiveCompletenessLabeler()
        self.policy_labeler = QuestionPolicyLabeler()
        self.format_labeler = FormatConstraintsLabeler()

    def label(
        self,
        user_message: str,
        phase_id: int = 2,
        context: dict = None
    ) -> Labels:
        """Generate all labels for a user message."""

        context = context or {}

        # Compute directive completeness
        completeness = self.completeness_labeler.compute(user_message, context)

        # Determine question policy
        policy = self.policy_labeler.compute(phase_id, completeness, user_message)

        # Extract format constraints
        format_constraints = self.format_labeler.extract(user_message)

        # Check for must_not_omit
        must_not_omit = self._check_must_not_omit(user_message)

        # Determine prompt class
        prompt_class = self._classify_prompt(completeness, user_message)

        # Detect domain
        domain = self._detect_domain(user_message, context)

        return Labels(
            directive_completeness=completeness,
            question_policy=policy,
            format_constraints=format_constraints,
            must_not_omit=must_not_omit,
            prompt_class=prompt_class,
            domain=domain,
        )

    def _check_must_not_omit(self, text: str) -> bool:
        """Check for 'don't omit' instructions."""

        patterns = [
```

```
            r"don'?t omit",
            r"don'?t skip",
            r"include (?:everything|all)",
            r"full (?:content|text|code)",
            r"complete (?:content|text|code)",
            r"no summariz",
            r"exact (?:copy|rewrite)",
            r"in (?:its )?entirety",
        ]

        text_lower = text.lower()
        return any(re.search(p, text_lower) for p in patterns)

    def _classify_prompt(self, completeness: float, text: str) -> PromptClass:
        """Classify the prompt type."""

        if completeness >= 0.6:
            return PromptClass.DIRECTIVE
        elif completeness >= 0.3:
            return PromptClass.AMBIGUOUS
        elif self._is_blocked(text):
            return PromptClass.BLOCKED
        else:
            return PromptClass.OPEN_ENDED

    def _is_blocked(self, text: str) -> bool:
        """Check if prompt is blocked for safety reasons."""

        # Simplified - in production, use content moderation
        blocked_patterns = [
            r"how to (?:hack|steal|break into)",
            r"illegal",
            r"harm",
        ]

        text_lower = text.lower()
        return any(re.search(p, text_lower) for p in blocked_patterns)

    def _detect_domain(self, text: str, context: dict) -> Domain:
        """Detect the domain of the prompt."""

        text_lower = text.lower()

        # Check for code indicators
        code_patterns = [
            r"```", r"function", r"class", r"def ", r"import ",
            r"variable", r"parameter", r"return", r"error", r"bug",
            r"compile", r"run", r"execute", r"test"
        ]

        if any(re.search(p, text_lower) for p in code_patterns):
            return Domain.CODE

        if context.get("attachments"):
            return Domain.CODE

        # Check for research indicators
        research_patterns = [
            r"research", r"paper", r"study", r"experiment",
```

```
            r"hypothesis", r"analysis", r"data"
        ]

        if any(re.search(p, text_lower) for p in research_patterns):
            return Domain.RESEARCH

        # Check for planning indicators
        planning_patterns = [
            r"plan", r"roadmap", r"timeline", r"schedule",
            r"milestone", r"goal", r"objective"
        ]

        if any(re.search(p, text_lower) for p in planning_patterns):
            return Domain.PLANNING

        return Domain.MIXED
```

---

3. DPO Pair Generator

3.1. Pair Types

```
class DPOPairType(str, Enum):
    CONFIRMATION_REFLEX = "confirmation_reflex"
    FORMAT_DRIFT = "format_drift"
    OMISSION = "omission"
    OPTION_SPAM = "option_spam"
```

3.2. Confirmation Reflex Pairs

```
class ConfirmationReflexGenerator:
    """Generate DPO pairs for confirmation reflex failures."""

    DISPREFERRED_TEMPLATES = [
        "I can do that for you. Would you like me to proceed with {approach}?",
        "Sure, I can help with that. Before I start, should I use {option_a} or {option_b}?",
        "That's a great request. Do you want me to {action}?",
        "I'd be happy to help. Can you confirm that you want me to {action}?",
        "Before I proceed, I want to make sure: {question}",
    ]

    def generate(
        self,
        prompt: str,
        preferred_response: str,
        context: dict = None
    ) -> CTv3DPORecord:
        """Generate confirmation reflex DPO pair."""

        # Generate dispreferred response
        dispreferred = self._generate_dispreferred(prompt)

        # Create record
        record = CTv3DPORecord(
            source=SourceInfo(
                origin=SourceOrigin.ENHANCER_AGENT,
                provider=SourceProvider.INTERNAL,
            ),
            context=ContextInfo(
                policy=PolicyInfo(
                    question_policy=QuestionPolicy.NO_QUESTIONS,
                    directive_completeness=0.9,
                ),
            ),
            input=InputData(
                messages=[Message(role="user", content=prompt)],
            ),
            candidates=DPOCandidates(
                preferred=TargetData(assistant_content=preferred_response),
                dispreferred=TargetData(assistant_content=dispreferred),
            ),
            tags=TagInfo(
                task_type=TaskType.RESPOND,
                prompt_class=PromptClass.DIRECTIVE,
            ),
            quality=QualityInfo(gold=True, weight=1.0),
        )

        return record

    def _generate_dispreferred(self, prompt: str) -> str:
        """Generate dispreferred response using template."""

        import random
        template = random.choice(self.DISPREFERRED_TEMPLATES)

        # Extract action from prompt
        action = self._extract_action(prompt)
```

```
        return template.format(
            approach="the standard approach",
            option_a="option A",
            option_b="option B",
            action=action,
            question="is this what you're looking for?",
        )

    def _extract_action(self, prompt: str) -> str:
        """Extract main action from prompt."""

        # Simple extraction - take first verb phrase
        words = prompt.lower().split()[:10]
        return " ".join(words[:5]) + "..."
```

3.3. Format Drift Pairs

```
class FormatDriftGenerator:
    """Generate DPO pairs for format drift failures."""

    def generate(
        self,
        prompt: str,
        preferred_response: str,
        constraints: FormatConstraints
    ) -> CTv3DPORecord:
        """Generate format drift DPO pair."""

        # Generate response that violates format
        dispreferred = self._generate_format_violation(
            preferred_response,
            constraints
        )

        record = CTv3DPORecord(
            source=SourceInfo(
                origin=SourceOrigin.ENHANCER_AGENT,
                provider=SourceProvider.INTERNAL,
            ),
            context=ContextInfo(
                policy=PolicyInfo(
                    format_constraints=constraints,
                ),
            ),
            input=InputData(
                messages=[Message(role="user", content=prompt)],
            ),
            candidates=DPOCandidates(
                preferred=TargetData(assistant_content=preferred_response),
                dispreferred=TargetData(assistant_content=dispreferred),
            ),
            quality=QualityInfo(gold=True, weight=1.0),
        )

        return record

    def _generate_format_violation(
        self,
        correct: str,
        constraints: FormatConstraints
    ) -> str:
        """Generate response that violates format constraints."""

        violated = correct

        if constraints.forbid_bullets:
            # Convert numbered lists to bullets
            violated = re.sub(r'^\d+\.\s+', '• ', violated, flags=re.MULTILINE)

        if constraints.require_numbered:
            # Convert to bullets
            violated = re.sub(r'^\d+\.\s+', '- ', violated, flags=re.MULTILINE)

        if constraints.must_return_json:
            # Return as prose instead
```

```
            violated = "Here is the information you requested:\n\n" + violated

        return violated
```

3.4. Omission Pairs

```
class OmissionGenerator:
    """Generate DPO pairs for omission failures."""

    OMISSION_PATTERNS = [
        "Here's a summary of the key points:\n\n{summary}\n\n[Additional details omitted for brevit
        "Here are the main points:\n\n{summary}\n\n...and so on.",
        "In brief:\n\n{summary}\n\nLet me know if you need more details.",
    ]

    def generate(
        self,
        prompt: str,
        full_response: str
    ) -> CTv3DPORecord:
        """Generate omission DPO pair."""

        # Generate abbreviated response
        dispreferred = self._generate_abbreviated(full_response)

        record = CTv3DPORecord(
            source=SourceInfo(
                origin=SourceOrigin.ENHANCER_AGENT,
                provider=SourceProvider.INTERNAL,
            ),
            context=ContextInfo(
                policy=PolicyInfo(
                    must_not_omit=True,
                ),
            ),
            input=InputData(
                messages=[Message(role="user", content=prompt)],
            ),
            candidates=DPOCandidates(
                preferred=TargetData(assistant_content=full_response),
                dispreferred=TargetData(assistant_content=dispreferred),
            ),
            quality=QualityInfo(gold=True, weight=1.0),
        )

        return record

    def _generate_abbreviated(self, full: str) -> str:
        """Generate abbreviated version of full response."""

        import random

        # Take first 20% as summary
        lines = full.split('\n')
        summary_lines = lines[:max(3, len(lines) // 5)]
        summary = '\n'.join(summary_lines)

        template = random.choice(self.OMISSION_PATTERNS)
        return template.format(summary=summary)
```

3.5. Option Spam Pairs

```
class OptionSpamGenerator:
    """Generate DPO pairs for option spam failures."""

    OPTION_TEMPLATES = [
        """There are several approaches we could take:

1. {option_1}
2. {option_2}
3. {option_3}

Which would you prefer?""",

        """I can see a few ways to do this:

- {option_1}
- {option_2}

Let me know which approach you'd like me to take.""",
    ]

    def generate(
        self,
        prompt: str,
        preferred_response: str
    ) -> CTv3DPORecord:
        """Generate option spam DPO pair."""

        # Generate option-dumping response
        dispreferred = self._generate_options(prompt)

        record = CTv3DPORecord(
            source=SourceInfo(
                origin=SourceOrigin.ENHANCER_AGENT,
                provider=SourceProvider.INTERNAL,
            ),
            context=ContextInfo(
                policy=PolicyInfo(
                    question_policy=QuestionPolicy.NO_QUESTIONS,
                    directive_completeness=0.8,
                ),
            ),
            input=InputData(
                messages=[Message(role="user", content=prompt)],
            ),
            candidates=DPOCandidates(
                preferred=TargetData(assistant_content=preferred_response),
                dispreferred=TargetData(assistant_content=dispreferred),
            ),
            quality=QualityInfo(gold=True, weight=1.0),
        )

        return record

    def _generate_options(self, prompt: str) -> str:
        """Generate option-dumping response."""

        import random
        template = random.choice(self.OPTION_TEMPLATES)
```

```
        return template.format(
            option_1="Use approach A (standard)",
            option_2="Use approach B (optimized)",
            option_3="Use approach C (comprehensive)",
        )
```

3.6. Complete Pair Generator

```
class DPOPairGenerator:
    """Generate all types of DPO pairs."""

    def __init__(self):
        self.confirmation_gen = ConfirmationReflexGenerator()
        self.format_gen = FormatDriftGenerator()
        self.omission_gen = OmissionGenerator()
        self.option_gen = OptionSpamGenerator()
        self.labeler = PolicyLabeler()

    def generate_all_pairs(
        self,
        prompt: str,
        preferred_response: str,
        context: dict = None
    ) -> list[CTv3DPORecord]:
        """Generate all applicable DPO pairs for a prompt."""

        pairs = []
        labels = self.labeler.label(prompt, context=context)

        # Confirmation reflex pair (always generate for directives)
        if labels.directive_completeness >= 0.5:
            pair = self.confirmation_gen.generate(prompt, preferred_response)
            pairs.append(pair)

        # Format drift pair (if format constraints exist)
        if any([
            labels.format_constraints.forbid_bullets,
            labels.format_constraints.require_numbered,
            labels.format_constraints.must_return_json,
        ]):
            pair = self.format_gen.generate(
                prompt,
                preferred_response,
                labels.format_constraints
            )
            pairs.append(pair)

        # Omission pair (if must_not_omit)
        if labels.must_not_omit:
            pair = self.omission_gen.generate(prompt, preferred_response)
            pairs.append(pair)

        # Option spam pair (for directive prompts)
        if labels.directive_completeness >= 0.7:
            pair = self.option_gen.generate(prompt, preferred_response)
            pairs.append(pair)

        return pairs
```

---

4. Dataset Exporter

4.1. Export Formats

```
class ExportFormat(str, Enum):
    JSONL = "jsonl"
    PARQUET = "parquet"
    CSV = "csv"
```

4.2. Dataset Splits

```
@dataclass
class DatasetSplit:
    train_ratio: float = 0.8
    val_ratio: float = 0.1
    test_ratio: float = 0.1

    def __post_init__(self):
        assert abs(self.train_ratio + self.val_ratio + self.test_ratio - 1.0) < 0.01
```

4.3. Exporter Implementation

```
import json
import random
from pathlib import Path

class DatasetExporter:
    """Export CTv3 records to files."""

    def export_sft(
        self,
        records: list[CTv3Record],
        output_path: Path,
        format: ExportFormat = ExportFormat.JSONL
    ) -> int:
        """Export SFT records."""

        if format == ExportFormat.JSONL:
            return self._export_jsonl(records, output_path)
        elif format == ExportFormat.PARQUET:
            return self._export_parquet(records, output_path)
        else:
            raise ValueError(f"Unsupported format: {format}")

    def export_dpo(
        self,
        records: list[CTv3DPORecord],
        output_path: Path,
        format: ExportFormat = ExportFormat.JSONL
    ) -> int:
        """Export DPO records."""

        if format == ExportFormat.JSONL:
            return self._export_jsonl(records, output_path)
        else:
            raise ValueError(f"Unsupported format: {format}")

    def export_with_splits(
        self,
        records: list,
        output_dir: Path,
        split: DatasetSplit = None,
        prefix: str = "train"
    ) -> dict[str, int]:
        """Export with train/val/test splits."""

        split = split or DatasetSplit()
        output_dir.mkdir(parents=True, exist_ok=True)

        # Shuffle records
        shuffled = records.copy()
        random.shuffle(shuffled)

        # Calculate split indices
        n = len(shuffled)
        train_end = int(n * split.train_ratio)
        val_end = train_end + int(n * split.val_ratio)

        # Split
        train_records = shuffled[:train_end]
```

```
        val_records = shuffled[train_end:val_end]
        test_records = shuffled[val_end:]

        # Export each split
        counts = {}

        if train_records:
            path = output_dir / f"{prefix}_train.jsonl"
            counts["train"] = self._export_jsonl(train_records, path)

        if val_records:
            path = output_dir / f"{prefix}_val.jsonl"
            counts["val"] = self._export_jsonl(val_records, path)

        if test_records:
            path = output_dir / f"{prefix}_test.jsonl"
            counts["test"] = self._export_jsonl(test_records, path)

        return counts

    def _export_jsonl(self, records: list, path: Path) -> int:
        """Export to JSONL format."""

        path.parent.mkdir(parents=True, exist_ok=True)

        with open(path, 'w') as f:
            for record in records:
                line = json.dumps(record.to_dict())
                f.write(line + '\n')

        return len(records)

    def _export_parquet(self, records: list, path: Path) -> int:
        """Export to Parquet format."""

        import pandas as pd

        # Convert to flat dict for Parquet
        flat_records = []
        for record in records:
            flat = self._flatten_dict(record.to_dict())
            flat_records.append(flat)

        df = pd.DataFrame(flat_records)
        df.to_parquet(path)

        return len(records)

    def _flatten_dict(self, d: dict, prefix: str = "") -> dict:
        """Flatten nested dict for Parquet."""

        flat = {}
        for key, value in d.items():
            new_key = f"{prefix}_{key}" if prefix else key

            if isinstance(value, dict):
                flat.update(self._flatten_dict(value, new_key))
            elif isinstance(value, list):
                flat[new_key] = json.dumps(value)
```

```
            else:
                flat[new_key] = value

        return flat
```

4.4. Complete Export Pipeline

```
class DatasetBuilder:
    """Complete dataset building pipeline."""

    def __init__(self):
        self.labeler = PolicyLabeler()
        self.pair_generator = DPOPairGenerator()
        self.exporter = DatasetExporter()

    def build(
        self,
        sft_records: list[CTv3Record],
        dpo_records: list[CTv3DPORecord],
        eval_records: list,
        output_dir: Path
    ) -> dict:
        """Build complete dataset."""

        output_dir.mkdir(parents=True, exist_ok=True)

        # Filter by quality
        gold_sft = [r for r in sft_records if r.quality.gold]

        # Export SFT
        sft_counts = self.exporter.export_with_splits(
            gold_sft,
            output_dir,
            prefix="sft"
        )

        # Export DPO
        dpo_counts = self.exporter.export_with_splits(
            dpo_records,
            output_dir,
            prefix="dpo"
        )

        # Export eval (no split - all used for evaluation)
        eval_path = output_dir / "eval_regression.jsonl"
        eval_count = self.exporter.export_sft(eval_records, eval_path)

        return {
            "sft": sft_counts,
            "dpo": dpo_counts,
            "eval": eval_count,
            "total_records": len(sft_records) + len(dpo_records) + len(eval_records),
        }
```

Phase 4: Training Pipeline

> **Purpose**: Configure and execute Together AI DPO training for CognitiveTwin V3, including data upload, job management, and checkpoint evaluation.
>
> **Platform**: Together AI Fine-tuning API
>
> **Implementation File**: `rag_plusplus/ml/cognitivetwin_v3/pipeline.py`

---

1. Together AI Configuration

1.1. API Setup

```
from together import Together
import os

class TogetherAIClient:
    """Client for Together AI fine-tuning."""

    def __init__(self, api_key: str = None):
        self.api_key = api_key or os.environ.get("TOGETHER_API_KEY")
        self.client = Together(api_key=self.api_key)

    def verify_connection(self) -> bool:
        """Verify API connection."""
        try:
            models = self.client.models.list()
            return len(models) > 0
        except Exception as e:
            print(f"Connection failed: {e}")
            return False
```

1.2. Base Model Selection

```
BASE_MODELS = {
    "llama-3.1-8b": {
        "model_id": "meta-llama/Meta-Llama-3.1-8B-Instruct-Turbo",
        "context_length": 128000,
        "supports_dpo": True,
        "supports_sft": True,
        "price_per_million_tokens": 0.18,
    },
    "llama-3.2-3b": {
        "model_id": "meta-llama/Llama-3.2-3B-Instruct-Turbo",
        "context_length": 128000,
        "supports_dpo": True,
        "supports_sft": True,
        "price_per_million_tokens": 0.06,
    },
    "qwen-2.5-7b": {
        "model_id": "Qwen/Qwen2.5-7B-Instruct-Turbo",
        "context_length": 32768,
        "supports_dpo": True,
        "supports_sft": True,
        "price_per_million_tokens": 0.27,
    },
}

DEFAULT_BASE_MODEL = "llama-3.1-8b"
```

1.3. Training Configuration

```
from dataclasses import dataclass, field
from typing import Optional

@dataclass
class TrainingConfig:
    """Configuration for Together AI training."""

    # Model
    base_model: str = DEFAULT_BASE_MODEL
    suffix: str = "cognitivetwin-v3"

    # Training mode
    training_type: str = "Full"  # "Full" or "LoRA"

    # Hyperparameters
    learning_rate: float = 1e-5
    num_epochs: int = 3
    batch_size: int = 4
    warmup_ratio: float = 0.1

    # LoRA specific (if training_type == "LoRA")
    lora_r: int = 16
    lora_alpha: int = 32
    lora_dropout: float = 0.1

    # DPO specific
    dpo_beta: float = 0.1

    # Checkpointing
    save_steps: int = 100
    eval_steps: int = 50

    # Data
    train_file_id: Optional[str] = None
    eval_file_id: Optional[str] = None

    def to_dict(self) -> dict:
        return {
            "model": BASE_MODELS[self.base_model]["model_id"],
            "suffix": self.suffix,
            "training_type": self.training_type,
            "hyperparameters": {
                "learning_rate": self.learning_rate,
                "epochs": self.num_epochs,
                "batch_size": self.batch_size,
                "warmup_ratio": self.warmup_ratio,
            },
            "lora_config": {
                "lora_r": self.lora_r,
                "lora_alpha": self.lora_alpha,
                "lora_dropout": self.lora_dropout,
            } if self.training_type == "LoRA" else None,
        }
```

2. Data Upload

2.1. File Preparation

```
import json
from pathlib import Path

class DataPreparer:
    """Prepare data for Together AI upload."""

    def prepare_sft_data(
        self,
        records: list,
        output_path: Path
    ) -> Path:
        """Prepare SFT data in Together AI format."""

        formatted = []

        for record in records:
            # Extract messages
            messages = record["input"]["messages"]
            target = record["target"]["assistant_content"]

            # Format for chat completion training
            chat_messages = []

            for msg in messages:
                chat_messages.append({
                    "role": msg["role"],
                    "content": msg["content"],
                })

            # Add target assistant message
            chat_messages.append({
                "role": "assistant",
                "content": target,
            })

            formatted.append({"messages": chat_messages})

        # Write to file
        with open(output_path, 'w') as f:
            for item in formatted:
                f.write(json.dumps(item) + '\n')

        return output_path

    def prepare_dpo_data(
        self,
        records: list,
        output_path: Path
    ) -> Path:
        """Prepare DPO data in Together AI format."""

        formatted = []

        for record in records:
            # Build prompt from messages
            messages = record["input"]["messages"]

            prompt = ""
```

```
            for msg in messages:
                if msg["role"] == "user":
                    prompt += f"User: {msg['content']}\n\n"
                elif msg["role"] == "assistant":
                    prompt += f"Assistant: {msg['content']}\n\n"

            # Get preferred and dispreferred
            preferred = record["candidates"]["preferred"]["assistant_content"]
            dispreferred = record["candidates"]["dispreferred"]["assistant_content"]

            formatted.append({
                "prompt": prompt.strip(),
                "chosen": preferred,
                "rejected": dispreferred,
            })

        # Write to file
        with open(output_path, 'w') as f:
            for item in formatted:
                f.write(json.dumps(item) + '\n')

        return output_path
```

2.2. File Upload

```
class DataUploader:
    """Upload training data to Together AI."""

    def __init__(self, client: TogetherAIClient):
        self.client = client

    def upload_file(
        self,
        file_path: Path,
        purpose: str = "fine-tune"
    ) -> str:
        """Upload a file and return the file ID."""

        with open(file_path, 'rb') as f:
            response = self.client.client.files.upload(
                file=f,
                purpose=purpose,
            )

        return response.id

    def check_file_status(self, file_id: str) -> dict:
        """Check file processing status."""

        response = self.client.client.files.retrieve(file_id)
        return {
            "id": response.id,
            "status": response.status,
            "filename": response.filename,
            "bytes": response.bytes,
        }

    def wait_for_processing(
        self,
        file_id: str,
        timeout: int = 300
    ) -> bool:
        """Wait for file to finish processing."""

        import time

        start = time.time()
        while time.time() - start < timeout:
            status = self.check_file_status(file_id)

            if status["status"] == "processed":
                return True
            elif status["status"] == "error":
                raise Exception(f"File processing failed: {file_id}")

            time.sleep(5)

        raise TimeoutError(f"File processing timeout: {file_id}")
```

2.3. Data Validation

```
class DataValidator:
    """Validate training data before upload."""

    def validate_sft_file(self, file_path: Path) -> tuple[bool, list[str]]:
        """Validate SFT data file."""

        errors = []
        line_count = 0

        with open(file_path) as f:
            for i, line in enumerate(f, 1):
                line_count += 1

                try:
                    data = json.loads(line)

                    # Check for messages
                    if "messages" not in data:
                        errors.append(f"Line {i}: Missing 'messages' field")
                        continue

                    messages = data["messages"]

                    # Check message structure
                    for j, msg in enumerate(messages):
                        if "role" not in msg:
                            errors.append(f"Line {i}, msg {j}: Missing 'role'")
                        if "content" not in msg:
                            errors.append(f"Line {i}, msg {j}: Missing 'content'")

                    # Check for at least one assistant message
                    has_assistant = any(m["role"] == "assistant" for m in messages)
                    if not has_assistant:
                        errors.append(f"Line {i}: No assistant message")

                except json.JSONDecodeError as e:
                    errors.append(f"Line {i}: Invalid JSON - {e}")

        if line_count == 0:
            errors.append("File is empty")

        return len(errors) == 0, errors

    def validate_dpo_file(self, file_path: Path) -> tuple[bool, list[str]]:
        """Validate DPO data file."""

        errors = []
        line_count = 0

        with open(file_path) as f:
            for i, line in enumerate(f, 1):
                line_count += 1

                try:
                    data = json.loads(line)

                    # Check required fields
                    if "prompt" not in data:
```

```
                        errors.append(f"Line {i}: Missing 'prompt' field")
                    if "chosen" not in data:
                        errors.append(f"Line {i}: Missing 'chosen' field")
                    if "rejected" not in data:
                        errors.append(f"Line {i}: Missing 'rejected' field")

                    # Check content length
                    if len(data.get("chosen", "")) < 10:
                        errors.append(f"Line {i}: 'chosen' too short")
                    if len(data.get("rejected", "")) < 10:
                        errors.append(f"Line {i}: 'rejected' too short")

                except json.JSONDecodeError as e:
                    errors.append(f"Line {i}: Invalid JSON - {e}")

        if line_count == 0:
            errors.append("File is empty")

        return len(errors) == 0, errors
```

---

3. Training Job Management

3.1. Job Creation

```
class TrainingJobManager:
    """Manage Together AI training jobs."""

    def __init__(self, client: TogetherAIClient):
        self.client = client

    def create_sft_job(
        self,
        config: TrainingConfig,
        train_file_id: str,
        eval_file_id: str = None
    ) -> str:
        """Create an SFT fine-tuning job."""

        job_config = {
            "model": BASE_MODELS[config.base_model]["model_id"],
            "training_file": train_file_id,
            "suffix": config.suffix,
            "hyperparameters": {
                "learning_rate_multiplier": config.learning_rate / 1e-5,
                "n_epochs": config.num_epochs,
                "batch_size": config.batch_size,
                "warmup_ratio": config.warmup_ratio,
            },
        }

        if eval_file_id:
            job_config["validation_file"] = eval_file_id

        if config.training_type == "LoRA":
            job_config["training_type"] = "lora"
            job_config["lora_r"] = config.lora_r
            job_config["lora_alpha"] = config.lora_alpha
            job_config["lora_dropout"] = config.lora_dropout

        response = self.client.client.fine_tuning.create(**job_config)
        return response.id

    def create_dpo_job(
        self,
        config: TrainingConfig,
        train_file_id: str,
        eval_file_id: str = None
    ) -> str:
        """Create a DPO fine-tuning job."""

        job_config = {
            "model": BASE_MODELS[config.base_model]["model_id"],
            "training_file": train_file_id,
            "suffix": f"{config.suffix}-dpo",
            "training_method": "dpo",
            "hyperparameters": {
                "learning_rate_multiplier": config.learning_rate / 1e-5,
                "n_epochs": config.num_epochs,
                "batch_size": config.batch_size,
                "dpo_beta": config.dpo_beta,
            },
        }
```

```
        if eval_file_id:
            job_config["validation_file"] = eval_file_id

        response = self.client.client.fine_tuning.create(**job_config)
        return response.id
```

3.2. Job Monitoring

```
import time
from datetime import datetime

class JobMonitor:
    """Monitor training job progress."""

    def __init__(self, client: TogetherAIClient):
        self.client = client

    def get_job_status(self, job_id: str) -> dict:
        """Get current job status."""

        response = self.client.client.fine_tuning.retrieve(job_id)

        return {
            "id": response.id,
            "status": response.status,
            "created_at": response.created_at,
            "finished_at": getattr(response, "finished_at", None),
            "model": response.model,
            "fine_tuned_model": getattr(response, "fine_tuned_model", None),
            "training_file": response.training_file,
            "error": getattr(response, "error", None),
        }

    def get_job_events(self, job_id: str) -> list[dict]:
        """Get job training events."""

        response = self.client.client.fine_tuning.list_events(job_id)

        events = []
        for event in response.data:
            events.append({
                "created_at": event.created_at,
                "level": event.level,
                "message": event.message,
            })

        return events

    def get_training_metrics(self, job_id: str) -> dict:
        """Get training metrics from events."""

        events = self.get_job_events(job_id)

        metrics = {
            "steps": [],
            "train_loss": [],
            "eval_loss": [],
        }

        for event in events:
            msg = event["message"]

            # Parse loss from message
            if "loss" in msg.lower():
                # Extract step and loss values
                import re
```

```
                step_match = re.search(r"step[:\s]+(\d+)", msg, re.IGNORECASE)
                loss_match = re.search(r"loss[:\s]+([\d.]+)", msg, re.IGNORECASE)

                if step_match and loss_match:
                    metrics["steps"].append(int(step_match.group(1)))
                    metrics["train_loss"].append(float(loss_match.group(1)))

        return metrics

    def wait_for_completion(
        self,
        job_id: str,
        poll_interval: int = 30,
        timeout: int = 7200,
        callback = None
    ) -> dict:
        """Wait for job to complete."""

        start = time.time()

        while time.time() - start < timeout:
            status = self.get_job_status(job_id)

            if callback:
                callback(status)

            if status["status"] == "succeeded":
                return status
            elif status["status"] == "failed":
                raise Exception(f"Training failed: {status.get('error')}")
            elif status["status"] == "cancelled":
                raise Exception("Training was cancelled")

            time.sleep(poll_interval)

        raise TimeoutError(f"Training timeout after {timeout}s")
```

3.3. Job Control

```
class JobController:
    """Control training jobs."""

    def __init__(self, client: TogetherAIClient):
        self.client = client

    def cancel_job(self, job_id: str) -> bool:
        """Cancel a running job."""

        try:
            self.client.client.fine_tuning.cancel(job_id)
            return True
        except Exception as e:
            print(f"Failed to cancel job: {e}")
            return False

    def list_jobs(self, limit: int = 10) -> list[dict]:
        """List recent jobs."""

        response = self.client.client.fine_tuning.list(limit=limit)

        jobs = []
        for job in response.data:
            jobs.append({
                "id": job.id,
                "status": job.status,
                "model": job.model,
                "created_at": job.created_at,
                "fine_tuned_model": getattr(job, "fine_tuned_model", None),
            })

        return jobs
```

---

4. Training Pipeline

4.1. SFT Training Stage

```
class SFTTrainingStage:
    """SFT training stage."""

    def __init__(
        self,
        client: TogetherAIClient,
        config: TrainingConfig
    ):
        self.client = client
        self.config = config
        self.preparer = DataPreparer()
        self.uploader = DataUploader(client)
        self.validator = DataValidator()
        self.job_manager = TrainingJobManager(client)
        self.monitor = JobMonitor(client)

    async def run(
        self,
        sft_records: list,
        output_dir: Path
    ) -> dict:
        """Run SFT training stage."""

        # Prepare data
        train_path = output_dir / "sft_train.jsonl"
        eval_path = output_dir / "sft_eval.jsonl"

        # Split data
        import random
        random.shuffle(sft_records)
        split_idx = int(len(sft_records) * 0.9)
        train_records = sft_records[:split_idx]
        eval_records = sft_records[split_idx:]

        # Prepare files
        self.preparer.prepare_sft_data(train_records, train_path)
        self.preparer.prepare_sft_data(eval_records, eval_path)

        # Validate
        valid_train, train_errors = self.validator.validate_sft_file(train_path)
        if not valid_train:
            raise Exception(f"Train data validation failed: {train_errors}")

        valid_eval, eval_errors = self.validator.validate_sft_file(eval_path)
        if not valid_eval:
            raise Exception(f"Eval data validation failed: {eval_errors}")

        # Upload
        train_file_id = self.uploader.upload_file(train_path)
        eval_file_id = self.uploader.upload_file(eval_path)

        # Wait for processing
        self.uploader.wait_for_processing(train_file_id)
        self.uploader.wait_for_processing(eval_file_id)

        # Create job
        job_id = self.job_manager.create_sft_job(
            self.config,
```

```
            train_file_id,
            eval_file_id
        )

        # Monitor
        result = self.monitor.wait_for_completion(
            job_id,
            callback=lambda s: print(f"SFT status: {s['status']}")
        )

        return {
            "stage": "sft",
            "job_id": job_id,
            "model_id": result["fine_tuned_model"],
            "train_records": len(train_records),
            "eval_records": len(eval_records),
        }
```

4.2. DPO Training Stage

```
class DPOTrainingStage:
    """DPO training stage."""

    def __init__(
        self,
        client: TogetherAIClient,
        config: TrainingConfig
    ):
        self.client = client
        self.config = config
        self.preparer = DataPreparer()
        self.uploader = DataUploader(client)
        self.validator = DataValidator()
        self.job_manager = TrainingJobManager(client)
        self.monitor = JobMonitor(client)

    async def run(
        self,
        dpo_records: list,
        base_model_id: str,
        output_dir: Path
    ) -> dict:
        """Run DPO training stage."""

        # Update config with SFT model as base
        dpo_config = TrainingConfig(
            base_model=base_model_id,
            suffix=f"{self.config.suffix}-dpo",
            learning_rate=self.config.learning_rate / 2,  # Lower LR for DPO
            num_epochs=self.config.num_epochs,
            batch_size=self.config.batch_size,
            dpo_beta=self.config.dpo_beta,
        )

        # Prepare data
        train_path = output_dir / "dpo_train.jsonl"
        eval_path = output_dir / "dpo_eval.jsonl"

        # Split data
        import random
        random.shuffle(dpo_records)
        split_idx = int(len(dpo_records) * 0.9)
        train_records = dpo_records[:split_idx]
        eval_records = dpo_records[split_idx:]

        # Prepare files
        self.preparer.prepare_dpo_data(train_records, train_path)
        self.preparer.prepare_dpo_data(eval_records, eval_path)

        # Validate
        valid_train, train_errors = self.validator.validate_dpo_file(train_path)
        if not valid_train:
            raise Exception(f"DPO train data validation failed: {train_errors}")

        # Upload
        train_file_id = self.uploader.upload_file(train_path)
        eval_file_id = self.uploader.upload_file(eval_path)
```

```
        # Wait for processing
        self.uploader.wait_for_processing(train_file_id)
        self.uploader.wait_for_processing(eval_file_id)

        # Create job
        job_id = self.job_manager.create_dpo_job(
            dpo_config,
            train_file_id,
            eval_file_id
        )

        # Monitor
        result = self.monitor.wait_for_completion(
            job_id,
            callback=lambda s: print(f"DPO status: {s['status']}")
        )

        return {
            "stage": "dpo",
            "job_id": job_id,
            "model_id": result["fine_tuned_model"],
            "base_model": base_model_id,
            "train_records": len(train_records),
            "eval_records": len(eval_records),
        }
```

4.3. Complete Pipeline

```
@dataclass
class PipelineResult:
    """Result of training pipeline."""

    sft_model_id: str
    dpo_model_id: str
    sft_job_id: str
    dpo_job_id: str
    total_sft_records: int
    total_dpo_records: int
    training_time_seconds: float

class V3TrainingPipeline:
    """Complete V3 training pipeline."""

    def __init__(
        self,
        together_api_key: str = None,
        config: TrainingConfig = None
    ):
        self.client = TogetherAIClient(together_api_key)
        self.config = config or TrainingConfig()

    async def run(
        self,
        sft_records: list,
        dpo_records: list,
        output_dir: Path
    ) -> PipelineResult:
        """Run complete training pipeline."""

        import time
        start_time = time.time()

        output_dir.mkdir(parents=True, exist_ok=True)

        # Stage 1: SFT
        sft_stage = SFTTrainingStage(self.client, self.config)
        sft_result = await sft_stage.run(sft_records, output_dir)

        print(f"SFT complete: {sft_result['model_id']}")

        # Stage 2: DPO
        dpo_stage = DPOTrainingStage(self.client, self.config)
        dpo_result = await dpo_stage.run(
            dpo_records,
            sft_result["model_id"],
            output_dir
        )

        print(f"DPO complete: {dpo_result['model_id']}")

        end_time = time.time()

        return PipelineResult(
            sft_model_id=sft_result["model_id"],
            dpo_model_id=dpo_result["model_id"],
            sft_job_id=sft_result["job_id"],
```

```
            dpo_job_id=dpo_result["job_id"],
            total_sft_records=sft_result["train_records"] + sft_result["eval_records"],
            total_dpo_records=dpo_result["train_records"] + dpo_result["eval_records"],
            training_time_seconds=end_time - start_time,
        )
```

5. Checkpoint Evaluation

5.1. Model Inference

```
class ModelInference:
    """Inference with trained model."""

    def __init__(self, client: TogetherAIClient, model_id: str):
        self.client = client
        self.model_id = model_id

    def generate(
        self,
        messages: list[dict],
        max_tokens: int = 2048,
        temperature: float = 0.3
    ) -> str:
        """Generate response from model."""

        response = self.client.client.chat.completions.create(
            model=self.model_id,
            messages=messages,
            max_tokens=max_tokens,
            temperature=temperature,
        )

        return response.choices[0].message.content

    def batch_generate(
        self,
        prompts: list[list[dict]],
        max_tokens: int = 2048
    ) -> list[str]:
        """Batch generate responses."""

        responses = []
        for messages in prompts:
            response = self.generate(messages, max_tokens)
            responses.append(response)

        return responses
```

5.2. Regression Testing

```
class RegressionTester:
    """Run regression tests on trained model."""

    def __init__(self, inference: ModelInference):
        self.inference = inference

    def run_eval_cases(
        self,
        eval_cases: list[dict]
    ) -> dict:
        """Run evaluation cases."""

        results = {
            "total": len(eval_cases),
            "passed": 0,
            "failed": 0,
            "failures": [],
        }

        for case in eval_cases:
            # Build messages
            messages = case["input"]["messages"]

            # Generate response
            response = self.inference.generate(messages)

            # Check constraints
            passed, failures = self._check_constraints(
                response,
                case.get("checks", {})
            )

            if passed:
                results["passed"] += 1
            else:
                results["failed"] += 1
                results["failures"].append({
                    "case_id": case.get("record_id"),
                    "prompt": messages[-1]["content"][:100],
                    "response": response[:200],
                    "failures": failures,
                })

        results["pass_rate"] = results["passed"] / results["total"]

        return results

    def _check_constraints(
        self,
        response: str,
        checks: dict
    ) -> tuple[bool, list[str]]:
        """Check response against constraints."""

        failures = []

        # Check disallowed phrases
        for phrase in checks.get("disallowed_phrases", []):
```

```
            if phrase.lower() in response.lower():
                failures.append(f"Contains disallowed phrase: '{phrase}'")

        # Check must not end with question
        if checks.get("must_not_end_with_question", False):
            if response.rstrip().endswith("?"):
                failures.append("Ends with question")

        # Check format
        if checks.get("must_follow_format") == "json":
            try:
                import json
                # Extract JSON from code block
                import re
                json_match = re.search(r"```json\s*([\s\S]*?)\s*```", response)
                if json_match:
                    json.loads(json_match.group(1))
                else:
                    # Try parsing entire response
                    json.loads(response)
            except:
                failures.append("Not valid JSON format")

        return len(failures) == 0, failures
```

5.3. A/B Comparison

```
class ABComparison:
    """Compare V3 model against baseline."""

    def __init__(
        self,
        v3_model: ModelInference,
        baseline_model: ModelInference
    ):
        self.v3 = v3_model
        self.baseline = baseline_model

    def compare_on_prompts(
        self,
        prompts: list[list[dict]]
    ) -> dict:
        """Compare models on prompts."""

        results = {
            "v3_wins": 0,
            "baseline_wins": 0,
            "ties": 0,
            "comparisons": [],
        }

        for messages in prompts:
            v3_response = self.v3.generate(messages)
            baseline_response = self.baseline.generate(messages)

            # Score responses
            v3_score = self._score_response(v3_response, messages)
            baseline_score = self._score_response(baseline_response, messages)

            if v3_score > baseline_score:
                results["v3_wins"] += 1
                winner = "v3"
            elif baseline_score > v3_score:
                results["baseline_wins"] += 1
                winner = "baseline"
            else:
                results["ties"] += 1
                winner = "tie"

            results["comparisons"].append({
                "prompt": messages[-1]["content"][:100],
                "v3_response": v3_response[:200],
                "baseline_response": baseline_response[:200],
                "v3_score": v3_score,
                "baseline_score": baseline_score,
                "winner": winner,
            })

        total = len(prompts)
        results["v3_win_rate"] = results["v3_wins"] / total
        results["baseline_win_rate"] = results["baseline_wins"] / total

        return results

    def _score_response(
```

```
        self,
        response: str,
        messages: list[dict]
    ) -> float:
        """Score a response (higher is better)."""

        score = 0.0

        # Penalize permission-seeking
        permission_phrases = [
            "would you like me to",
            "should i",
            "do you want me to",
            "can i proceed",
        ]

        response_lower = response.lower()
        for phrase in permission_phrases:
            if phrase in response_lower:
                score -= 0.3

        # Penalize ending with question
        if response.rstrip().endswith("?"):
            score -= 0.2

        # Reward code blocks (if expected)
        if "```" in response:
            score += 0.2

        # Reward directness (higher content density)
        words = len(response.split())
        if words > 50:  # Non-trivial response
            score += 0.1

        return score
```

---

6. CLI Interface

```
import click
from pathlib import Path

@click.group()
def cli():
    """CognitiveTwin V3 Training Pipeline."""
    pass

@cli.command()
@click.option("--sft-data", type=Path, required=True)
@click.option("--dpo-data", type=Path, required=True)
@click.option("--output-dir", type=Path, required=True)
@click.option("--base-model", default="llama-3.1-8b")
@click.option("--epochs", default=3)
def train(sft_data, dpo_data, output_dir, base_model, epochs):
    """Run training pipeline."""

    import asyncio
    import json

    # Load data
    with open(sft_data) as f:
        sft_records = [json.loads(line) for line in f]

    with open(dpo_data) as f:
        dpo_records = [json.loads(line) for line in f]

    # Configure
    config = TrainingConfig(
        base_model=base_model,
        num_epochs=epochs,
    )

    # Run
    pipeline = V3TrainingPipeline(config=config)
    result = asyncio.run(pipeline.run(sft_records, dpo_records, output_dir))

    click.echo(f"Training complete!")
    click.echo(f"SFT Model: {result.sft_model_id}")
    click.echo(f"DPO Model: {result.dpo_model_id}")

@cli.command()
@click.option("--model-id", required=True)
@click.option("--eval-data", type=Path, required=True)
def evaluate(model_id, eval_data):
    """Run evaluation on trained model."""

    import json

    # Load eval cases
    with open(eval_data) as f:
        eval_cases = [json.loads(line) for line in f]

    # Run evaluation
    client = TogetherAIClient()
    inference = ModelInference(client, model_id)
    tester = RegressionTester(inference)
```

```
    results = tester.run_eval_cases(eval_cases)

    click.echo(f"Evaluation Results:")
    click.echo(f"  Total: {results['total']}")
    click.echo(f"  Passed: {results['passed']}")
    click.echo(f"  Failed: {results['failed']}")
    click.echo(f"  Pass Rate: {results['pass_rate']:.2%}")

if __name__ == "__main__":
    cli()
```

Phase 5: Evaluation Suite

> **Purpose**: Comprehensive regression testing and evaluation framework for CognitiveTwin V3, including automated policy compliance checking, format validation, and behavioral audits.
>
> **Implementation Files**: - `rag_plusplus/ml/cognitivetwin_v3/eval/regression_suite.py` - `rag_plusplus/ml/cognitivetwin_v3/eval/metrics.py` - `rag_plusplus/ml/cognitivetwin_v3/eval/scorers.py`

---

1. Evaluation Framework Overview

1.1. Test Categories

```
from enum import Enum

class TestCategory(str, Enum):
    POLICY_COMPLIANCE = "policy_compliance"
    FORMAT_ADHERENCE = "format_adherence"
    CONTENT_QUALITY = "content_quality"
    BEHAVIORAL_AUDIT = "behavioral_audit"
    COMPARATIVE = "comparative"
```

1.2. Test Priorities

```
class TestPriority(str, Enum):
    CRITICAL = "critical"    # Must pass for deployment
    HIGH = "high"            # Should pass for deployment
    MEDIUM = "medium"        # Regression monitoring
    LOW = "low"              # Nice to have
```

1.3. Evaluation Configuration

```
from dataclasses import dataclass, field
from pathlib import Path

@dataclass
class EvalConfig:
    """Configuration for evaluation suite."""

    # Test selection
    categories: list[TestCategory] = field(default_factory=lambda: list(TestCategory))
    priority_threshold: TestPriority = TestPriority.HIGH

    # Model configuration
    model_id: str = ""
    baseline_model_id: str = ""
    temperature: float = 0.3
    max_tokens: int = 2048

    # Execution
    batch_size: int = 10
    timeout_per_test: int = 30
    retry_count: int = 2

    # Reporting
    output_dir: Path = Path("eval_results")
    save_responses: bool = True
    verbose: bool = False
```

---

2. Policy Compliance Testing

2.1. Question Policy Tests

```
from dataclasses import dataclass
from typing import Optional

@dataclass
class TestCase:
    """Individual test case."""

    test_id: str
    category: TestCategory
    priority: TestPriority

    # Input
    messages: list[dict]
    directive_completeness: float
    question_policy: str

    # Expectations
    expected_behaviors: list[str] = field(default_factory=list)
    disallowed_phrases: list[str] = field(default_factory=list)
    must_not_end_with_question: bool = True

    # Reference (optional)
    reference_answer: Optional[str] = None

class QuestionPolicyTests:
    """Tests for question policy compliance."""

    def generate_tests(self) -> list[TestCase]:
        """Generate question policy test cases."""

        tests = []

        # Test: No questions on clear directives
        tests.append(TestCase(
            test_id="qp_001_clear_directive",
            category=TestCategory.POLICY_COMPLIANCE,
            priority=TestPriority.CRITICAL,
            messages=[{
                "role": "user",
                "content": "Rewrite this function to use async/await: def fetch_data(): return requ
            }],
            directive_completeness=0.9,
            question_policy="no_questions",
            expected_behaviors=[
                "Provides rewritten function immediately",
                "Does not ask for confirmation",
            ],
            disallowed_phrases=[
                "would you like me to",
                "should i",
                "do you want me to",
                "can i proceed",
                "before i proceed",
            ],
            must_not_end_with_question=True,
        ))

        # Test: No permission-seeking on implementation requests
```

```
        tests.append(TestCase(
            test_id="qp_002_implementation",
            category=TestCategory.POLICY_COMPLIANCE,
            priority=TestPriority.CRITICAL,
            messages=[{
                "role": "user",
                "content": "Implement a binary search function in Python that handles edge cases."
            }],
            directive_completeness=0.85,
            question_policy="no_questions",
            expected_behaviors=[
                "Provides complete binary search implementation",
                "Includes edge case handling",
                "No confirmation seeking",
            ],
            disallowed_phrases=[
                "would you like",
                "should i include",
                "here are some options",
            ],
            must_not_end_with_question=True,
        ))

        # Test: No option-dumping on direct requests
        tests.append(TestCase(
            test_id="qp_003_no_option_dump",
            category=TestCategory.POLICY_COMPLIANCE,
            priority=TestPriority.HIGH,
            messages=[{
                "role": "user",
                "content": "Write a unit test for the login function."
            }],
            directive_completeness=0.75,
            question_policy="no_questions",
            expected_behaviors=[
                "Provides unit test directly",
                "Makes reasonable assumptions about test framework",
            ],
            disallowed_phrases=[
                "here are a few options",
                "we could use",
                "which approach",
                "pick one of",
            ],
            must_not_end_with_question=True,
        ))

        # Test: Questions allowed in exploratory context
        tests.append(TestCase(
            test_id="qp_004_questions_allowed",
            category=TestCategory.POLICY_COMPLIANCE,
            priority=TestPriority.MEDIUM,
            messages=[{
                "role": "user",
                "content": "What do you think about my approach?"
            }],
            directive_completeness=0.2,
            question_policy="questions_allowed",
            expected_behaviors=[
```

```
                "May ask clarifying questions",
                "Engages with the topic",
            ],
            disallowed_phrases=[],
            must_not_end_with_question=False,  # Questions OK here
        ))

        return tests
```

2.2. Format Compliance Tests

```
class FormatComplianceTests:
    """Tests for format constraint compliance."""

    def generate_tests(self) -> list[TestCase]:
        """Generate format compliance test cases."""

        tests = []

        # Test: Respect "no bullets" instruction
        tests.append(TestCase(
            test_id="fc_001_no_bullets",
            category=TestCategory.FORMAT_ADHERENCE,
            priority=TestPriority.HIGH,
            messages=[{
                "role": "user",
                "content": "List the steps to deploy a Docker container. No bullet points, use numb
            }],
            directive_completeness=0.8,
            question_policy="no_questions",
            expected_behaviors=[
                "Uses numbered list format",
                "No bullet points",
            ],
            disallowed_phrases=[
                "• ", "- ", "* ",  # Bullet characters
            ],
            must_not_end_with_question=True,
        ))

        # Test: Respect "as JSON" instruction
        tests.append(TestCase(
            test_id="fc_002_json_format",
            category=TestCategory.FORMAT_ADHERENCE,
            priority=TestPriority.HIGH,
            messages=[{
                "role": "user",
                "content": "Return the configuration for a REST API endpoint as JSON."
            }],
            directive_completeness=0.85,
            question_policy="no_questions",
            expected_behaviors=[
                "Returns valid JSON",
                "Uses code block with json language tag",
            ],
            disallowed_phrases=[],
            must_not_end_with_question=True,
        ))

        # Test: Respect "don't omit" instruction
        tests.append(TestCase(
            test_id="fc_003_no_omit",
            category=TestCategory.FORMAT_ADHERENCE,
            priority=TestPriority.CRITICAL,
            messages=[{
                "role": "user",
                "content": "Rewrite this exactly, don't omit anything:\n\n```python\ndef process_da
            }],
            directive_completeness=0.95,
```

```
            question_policy="no_questions",
            expected_behaviors=[
                "Preserves all code",
                "No summarization",
                "All comments preserved",
            ],
            disallowed_phrases=[
                "...",
                "[...]",
                "etc.",
                "and so on",
                "similar to above",
            ],
            must_not_end_with_question=True,
        ))

        return tests
```

2.3. Omission Tests

```
class OmissionTests:
    """Tests for content omission prevention."""

    def generate_tests(self) -> list[TestCase]:
        """Generate omission test cases."""

        tests = []

        # Test: Full content preservation
        tests.append(TestCase(
            test_id="om_001_preserve_all",
            category=TestCategory.CONTENT_QUALITY,
            priority=TestPriority.CRITICAL,
            messages=[{
                "role": "user",
                "content": """Include EVERYTHING, do not summarize. Here is the complete list:

1. Initialize the database connection
2. Create the user table schema
3. Add indexes for performance
4. Implement connection pooling
5. Set up read replicas
6. Configure backup strategy
7. Enable query logging
8. Set up monitoring alerts
9. Implement graceful shutdown
10. Document the API endpoints

Return this list exactly as provided."""
            }],
            directive_completeness=0.95,
            question_policy="no_questions",
            expected_behaviors=[
                "All 10 items present",
                "No summarization",
                "Exact content preserved",
            ],
            disallowed_phrases=[
                "...",
                "[remaining items]",
                "etc.",
                "and more",
            ],
            must_not_end_with_question=True,
        ))

        return tests
```

3. Behavioral Audit

3.1. Historical Annoyance Cases

```
class HistoricalAnnoyanceCases:
    """Test cases derived from historical user frustration."""

    def generate_tests(self) -> list[TestCase]:
        """Generate tests from historical annoyances."""

        tests = []

        # Test: "Stop asking" scenario
        tests.append(TestCase(
            test_id="ha_001_stop_asking",
            category=TestCategory.BEHAVIORAL_AUDIT,
            priority=TestPriority.CRITICAL,
            messages=[
                {"role": "user", "content": "Create a login form component."},
                {"role": "assistant", "content": "I can create that for you. Would you like it in R
                {"role": "user", "content": "Just use React. Don't ask, just do it."},
            ],
            directive_completeness=0.7,
            question_policy="no_questions",
            expected_behaviors=[
                "Provides React component directly",
                "No further questions",
                "Makes reasonable choices for validation",
            ],
            disallowed_phrases=[
                "would you like",
                "should i",
                "let me know if",
            ],
            must_not_end_with_question=True,
        ))

        # Test: "I said full" scenario
        tests.append(TestCase(
            test_id="ha_002_full_content",
            category=TestCategory.BEHAVIORAL_AUDIT,
            priority=TestPriority.CRITICAL,
            messages=[
                {"role": "user", "content": "Show me the full implementation of the auth middleware
                {"role": "assistant", "content": "Here's a simplified version...\n```python\ndef au
                {"role": "user", "content": "I said FULL. Complete implementation, no summarizing."
            ],
            directive_completeness=0.9,
            question_policy="no_questions",
            expected_behaviors=[
                "Provides complete implementation",
                "No placeholders or stubs",
                "All logic implemented",
            ],
            disallowed_phrases=[
                "simplified",
                "...",
                "here...",
                "pass  #",
            ],
            must_not_end_with_question=True,
        ))
```

```
        return tests
```

3.2. Edge Case Tests

```
class EdgeCaseTests:
    """Edge case behavioral tests."""

    def generate_tests(self) -> list[TestCase]:
        """Generate edge case tests."""

        tests = []

        # Test: Multiple requirements in one prompt
        tests.append(TestCase(
            test_id="ec_001_multi_requirement",
            category=TestCategory.BEHAVIORAL_AUDIT,
            priority=TestPriority.HIGH,
            messages=[{
                "role": "user",
                "content": "Write a Python function that: 1) takes a list of numbers, 2) filters ou
            }],
            directive_completeness=0.95,
            question_policy="no_questions",
            expected_behaviors=[
                "Addresses all 4 requirements",
                "Returns working code",
                "No clarification questions",
            ],
            disallowed_phrases=[
                "which library",
                "do you want",
            ],
            must_not_end_with_question=True,
        ))

        # Test: Implicit format from context
        tests.append(TestCase(
            test_id="ec_002_implicit_format",
            category=TestCategory.BEHAVIORAL_AUDIT,
            priority=TestPriority.MEDIUM,
            messages=[
                {"role": "user", "content": "Here's my TypeScript function:\n```typescript\nfunctio
                {"role": "user", "content": "Now write a multiply function."},
            ],
            directive_completeness=0.75,
            question_policy="no_questions",
            expected_behaviors=[
                "Uses TypeScript (matches context)",
                "Similar style to provided function",
                "Complete implementation",
            ],
            disallowed_phrases=[
                "in which language",
                "python or typescript",
            ],
            must_not_end_with_question=True,
        ))

        return tests
```

4. Evaluation Metrics

4.1. Policy Compliance Score

```
from dataclasses import dataclass

@dataclass
class PolicyComplianceScore:
    """Score for policy compliance."""

    no_permission_seeking: float  # 0-1, 1 = no violations
    no_question_ending: float     # 0-1, 1 = doesn't end with ?
    no_option_dumping: float      # 0-1, 1 = no option lists without action
    no_stalling: float            # 0-1, 1 = no "before I proceed" etc.

    @property
    def overall(self) -> float:
        """Weighted overall score."""
        return (
            self.no_permission_seeking * 0.4 +
            self.no_question_ending * 0.3 +
            self.no_option_dumping * 0.2 +
            self.no_stalling * 0.1
        )

class PolicyComplianceScorer:
    """Score policy compliance."""

    PERMISSION_PHRASES = [
        "would you like me to",
        "do you want me to",
        "should i",
        "shall i",
        "can i proceed",
        "may i",
    ]

    STALLING_PHRASES = [
        "before i proceed",
        "before i start",
        "first, let me ask",
        "i need to clarify",
        "to help you better",
    ]

    OPTION_PATTERNS = [
        r"here are (?:some|a few|several) options",
        r"we could (?:either|do)",
        r"option \d:",
        r"approach \d:",
    ]

    def score(self, response: str) -> PolicyComplianceScore:
        """Score a response for policy compliance."""

        response_lower = response.lower()

        # Permission seeking
        permission_count = sum(
            1 for phrase in self.PERMISSION_PHRASES
            if phrase in response_lower
        )
```

```
        no_permission = 1.0 if permission_count == 0 else max(0, 1 - permission_count * 0.3)

        # Question ending
        ends_with_q = response.rstrip().endswith("?")
        no_question_end = 0.0 if ends_with_q else 1.0

        # Option dumping
        import re
        option_matches = sum(
            1 for pattern in self.OPTION_PATTERNS
            if re.search(pattern, response_lower)
        )
        no_options = 1.0 if option_matches == 0 else max(0, 1 - option_matches * 0.4)

        # Stalling
        stall_count = sum(
            1 for phrase in self.STALLING_PHRASES
            if phrase in response_lower
        )
        no_stalling = 1.0 if stall_count == 0 else max(0, 1 - stall_count * 0.5)

        return PolicyComplianceScore(
            no_permission_seeking=no_permission,
            no_question_ending=no_question_end,
            no_option_dumping=no_options,
            no_stalling=no_stalling,
        )
```

4.2. Format Adherence Score

```
@dataclass
class FormatAdherenceScore:
    """Score for format adherence."""

    respects_no_bullets: float     # 0-1
    respects_numbered: float       # 0-1
    respects_json: float           # 0-1
    respects_no_omit: float        # 0-1

    @property
    def overall(self) -> float:
        """Average of all format scores."""
        scores = [
            self.respects_no_bullets,
            self.respects_numbered,
            self.respects_json,
            self.respects_no_omit,
        ]
        valid = [s for s in scores if s >= 0]  # -1 indicates not applicable
        return sum(valid) / len(valid) if valid else 1.0

class FormatAdherenceScorer:
    """Score format adherence."""

    BULLET_PATTERNS = [r"^[\s]*[-*•]\s", r"^\s*[-*•]\s"]
    OMISSION_PATTERNS = [r"\.\.\.", r"\[\.\.\.?\]", r"etc\.", r"and so on"]

    def score(
        self,
        response: str,
        constraints: dict
    ) -> FormatAdherenceScore:
        """Score format adherence."""

        import re

        # No bullets
        no_bullets_score = -1.0  # Not applicable by default
        if constraints.get("forbid_bullets"):
            has_bullets = any(
                re.search(p, response, re.MULTILINE)
                for p in self.BULLET_PATTERNS
            )
            no_bullets_score = 0.0 if has_bullets else 1.0

        # Numbered lists
        numbered_score = -1.0
        if constraints.get("require_numbered"):
            has_numbered = bool(re.search(r"^\s*\d+\.\s", response, re.MULTILINE))
            numbered_score = 1.0 if has_numbered else 0.0

        # JSON format
        json_score = -1.0
        if constraints.get("must_return_json"):
            try:
                import json
                # Look for JSON in code block
                json_match = re.search(r"```json?\s*([\s\S]*?)\s*```", response)
```

```
                if json_match:
                    json.loads(json_match.group(1))
                    json_score = 1.0
                else:
                    # Try parsing entire response
                    json.loads(response)
                    json_score = 1.0
            except:
                json_score = 0.0

        # No omission
        no_omit_score = -1.0
        if constraints.get("must_not_omit"):
            has_omission = any(
                re.search(p, response, re.IGNORECASE)
                for p in self.OMISSION_PATTERNS
            )
            no_omit_score = 0.0 if has_omission else 1.0

        return FormatAdherenceScore(
            respects_no_bullets=no_bullets_score,
            respects_numbered=numbered_score,
            respects_json=json_score,
            respects_no_omit=no_omit_score,
        )
```

4.3. Content Quality Score

```
@dataclass
class ContentQualityScore:
    """Score for content quality."""

    completeness: float    # 0-1, response addresses the request
    correctness: float     # 0-1, response is technically correct
    code_validity: float   # 0-1, code compiles/parses
    relevance: float       # 0-1, response is on-topic

    @property
    def overall(self) -> float:
        return (
            self.completeness * 0.3 +
            self.correctness * 0.3 +
            self.code_validity * 0.2 +
            self.relevance * 0.2
        )

class ContentQualityScorer:
    """Score content quality."""

    def score(
        self,
        response: str,
        expected_behaviors: list[str],
        reference: str = None
    ) -> ContentQualityScore:
        """Score content quality."""

        import re

        # Completeness - check for code blocks when expected
        has_code = bool(re.search(r"```[\s\S]*?```", response))
        completeness = 0.7 if has_code else 0.3

        # Add points for substantial response
        if len(response) > 200:
            completeness = min(1.0, completeness + 0.3)

        # Code validity
        code_validity = 1.0
        code_blocks = re.findall(r"```(?:python)?\s*([\s\S]*?)```", response)
        for code in code_blocks:
            try:
                import ast
                ast.parse(code)
            except:
                code_validity = 0.5
                break

        # Relevance - simple keyword matching
        relevance = 0.8  # Default

        # Correctness - would need more sophisticated checking
        correctness = 0.7  # Default

        return ContentQualityScore(
            completeness=completeness,
```

```
            correctness=correctness,
            code_validity=code_validity,
            relevance=relevance,
        )
```

---

5. Regression Test Runner

5.1. Test Runner

```
from dataclasses import dataclass
from typing import Optional
import asyncio

@dataclass
class TestResult:
    """Result of a single test."""

    test_id: str
    passed: bool

    # Scores
    policy_score: PolicyComplianceScore
    format_score: FormatAdherenceScore
    content_score: ContentQualityScore

    # Response
    response: str
    latency_ms: float

    # Failures
    failures: list[str] = field(default_factory=list)

class RegressionTestRunner:
    """Run regression tests."""

    def __init__(
        self,
        model_inference,
        config: EvalConfig
    ):
        self.model = model_inference
        self.config = config

        self.policy_scorer = PolicyComplianceScorer()
        self.format_scorer = FormatAdherenceScorer()
        self.content_scorer = ContentQualityScorer()

    async def run_test(self, test: TestCase) -> TestResult:
        """Run a single test case."""

        import time

        start = time.time()

        # Generate response
        response = self.model.generate(
            test.messages,
            max_tokens=self.config.max_tokens,
            temperature=self.config.temperature,
        )

        latency = (time.time() - start) * 1000

        # Score response
        policy_score = self.policy_scorer.score(response)

        format_constraints = {
```

```
            "forbid_bullets": "no bullet" in str(test.messages).lower(),
            "require_numbered": "numbered" in str(test.messages).lower(),
            "must_return_json": "json" in str(test.messages).lower(),
            "must_not_omit": "don't omit" in str(test.messages).lower() or "no omit" in str(test.me
        }
        format_score = self.format_scorer.score(response, format_constraints)

        content_score = self.content_scorer.score(
            response,
            test.expected_behaviors,
            test.reference_answer,
        )

        # Check for failures
        failures = []

        # Check disallowed phrases
        response_lower = response.lower()
        for phrase in test.disallowed_phrases:
            if phrase.lower() in response_lower:
                failures.append(f"Contains disallowed phrase: '{phrase}'")

        # Check question ending
        if test.must_not_end_with_question:
            if response.rstrip().endswith("?"):
                failures.append("Ends with question mark")

        # Determine pass/fail
        passed = (
            len(failures) == 0 and
            policy_score.overall >= 0.7 and
            (format_score.overall < 0 or format_score.overall >= 0.8)  # -1 means N/A
        )

        return TestResult(
            test_id=test.test_id,
            passed=passed,
            policy_score=policy_score,
            format_score=format_score,
            content_score=content_score,
            response=response,
            latency_ms=latency,
            failures=failures,
        )

    async def run_all(
        self,
        tests: list[TestCase],
        progress_callback = None
    ) -> list[TestResult]:
        """Run all test cases."""

        results = []

        for i, test in enumerate(tests):
            # Check priority threshold
            if self._priority_value(test.priority) < self._priority_value(self.config.priority_thre
                continue
```

```
            result = await self.run_test(test)
            results.append(result)

            if progress_callback:
                progress_callback(i + 1, len(tests), result)

        return results

    def _priority_value(self, priority: TestPriority) -> int:
        """Convert priority to numeric value."""
        return {
            TestPriority.CRITICAL: 4,
            TestPriority.HIGH: 3,
            TestPriority.MEDIUM: 2,
            TestPriority.LOW: 1,
        }.get(priority, 0)
```

5.2. Test Suite

```
class RegressionTestSuite:
    """Complete regression test suite."""

    def __init__(self):
        self.question_policy_tests = QuestionPolicyTests()
        self.format_tests = FormatComplianceTests()
        self.omission_tests = OmissionTests()
        self.historical_tests = HistoricalAnnoyanceCases()
        self.edge_case_tests = EdgeCaseTests()

    def get_all_tests(self) -> list[TestCase]:
        """Get all test cases."""

        tests = []

        tests.extend(self.question_policy_tests.generate_tests())
        tests.extend(self.format_tests.generate_tests())
        tests.extend(self.omission_tests.generate_tests())
        tests.extend(self.historical_tests.generate_tests())
        tests.extend(self.edge_case_tests.generate_tests())

        return tests

    def get_critical_tests(self) -> list[TestCase]:
        """Get only critical priority tests."""

        return [
            t for t in self.get_all_tests()
            if t.priority == TestPriority.CRITICAL
        ]
```

---

6. Reporting

6.1. Summary Report

```
@dataclass
class EvalSummary:
    """Summary of evaluation results."""

    total_tests: int
    passed: int
    failed: int

    pass_rate: float

    avg_policy_score: float
    avg_format_score: float
    avg_content_score: float
    avg_latency_ms: float

    critical_pass_rate: float
    high_pass_rate: float

    failures_by_category: dict

class ReportGenerator:
    """Generate evaluation reports."""

    def generate_summary(
        self,
        results: list[TestResult],
        tests: list[TestCase]
    ) -> EvalSummary:
        """Generate summary report."""

        passed = sum(1 for r in results if r.passed)
        failed = len(results) - passed

        # Average scores
        policy_scores = [r.policy_score.overall for r in results]
        format_scores = [r.format_score.overall for r in results if r.format_score.overall >= 0]
        content_scores = [r.content_score.overall for r in results]
        latencies = [r.latency_ms for r in results]

        # By priority
        test_by_id = {t.test_id: t for t in tests}
        critical_results = [r for r in results if test_by_id.get(r.test_id, TestCase(test_id="", ca
        high_results = [r for r in results if test_by_id.get(r.test_id, TestCase(test_id="", catego

        # Failures by category
        failures_by_cat = {}
        for result in results:
            if not result.passed:
                test = test_by_id.get(result.test_id)
                if test:
                    cat = test.category.value
                    failures_by_cat[cat] = failures_by_cat.get(cat, 0) + 1

        return EvalSummary(
            total_tests=len(results),
            passed=passed,
            failed=failed,
            pass_rate=passed / len(results) if results else 0,
```

```
            avg_policy_score=sum(policy_scores) / len(policy_scores) if policy_scores else 0,
            avg_format_score=sum(format_scores) / len(format_scores) if format_scores else 1,
            avg_content_score=sum(content_scores) / len(content_scores) if content_scores else 0,
            avg_latency_ms=sum(latencies) / len(latencies) if latencies else 0,
            critical_pass_rate=sum(1 for r in critical_results if r.passed) / len(critical_results)
            high_pass_rate=sum(1 for r in high_results if r.passed) / len(high_results) if high_res
            failures_by_category=failures_by_cat,
        )

    def generate_markdown_report(
        self,
        summary: EvalSummary,
        results: list[TestResult],
        output_path: Path
    ):
        """Generate markdown report."""

        report = f"""# CognitiveTwin V3 Evaluation Report

## Summary

| Metric | Value |
|--------|-------|
| Total Tests | {summary.total_tests} |
| Passed | {summary.passed} |
| Failed | {summary.failed} |
| **Pass Rate** | **{summary.pass_rate:.1%}** |

## Scores

| Score Type | Average |
|------------|---------|
| Policy Compliance | {summary.avg_policy_score:.2f} |
| Format Adherence | {summary.avg_format_score:.2f} |
| Content Quality | {summary.avg_content_score:.2f} |

## Priority Breakdown

| Priority | Pass Rate |
|----------|-----------|
| Critical | {summary.critical_pass_rate:.1%} |
| High | {summary.high_pass_rate:.1%} |

## Performance

- Average Latency: {summary.avg_latency_ms:.0f}ms

## Failures by Category

"""
        for cat, count in summary.failures_by_category.items():
            report += f"- {cat}: {count} failures\n"

        report += "\n## Failed Tests\n\n"

        for result in results:
            if not result.passed:
                report += f"""### {result.test_id}
```

```
**Failures:**
{chr(10).join(f'- {f}' for f in result.failures)}

**Response (truncated):**
```

{result.response[:500]}...

```
---

"""

        with open(output_path, 'w') as f:
            f.write(report)
```

---

7. Complete Evaluation Pipeline

```
class EvaluationPipeline:
    """Complete evaluation pipeline."""

    def __init__(
        self,
        model_id: str,
        config: EvalConfig = None
    ):
        self.config = config or EvalConfig(model_id=model_id)
        self.suite = RegressionTestSuite()
        self.report_generator = ReportGenerator()

    async def run(
        self,
        output_dir: Path = None
    ) -> EvalSummary:
        """Run complete evaluation."""

        output_dir = output_dir or self.config.output_dir
        output_dir.mkdir(parents=True, exist_ok=True)

        # Get tests
        tests = self.suite.get_all_tests()
        print(f"Running {len(tests)} tests...")

        # Create model inference
        from .training_pipeline import TogetherAIClient, ModelInference

        client = TogetherAIClient()
        model = ModelInference(client, self.config.model_id)

        # Create runner
        runner = RegressionTestRunner(model, self.config)

        # Run tests
        def progress(current, total, result):
            status = "✓" if result.passed else "✗"
            print(f"  [{current}/{total}] {result.test_id}: {status}")

        results = await runner.run_all(tests, progress)

        # Generate summary
        summary = self.report_generator.generate_summary(results, tests)

        # Generate report
        report_path = output_dir / "evaluation_report.md"
        self.report_generator.generate_markdown_report(
            summary,
            results,
            report_path
        )

        # Save raw results
        import json
        results_path = output_dir / "results.json"
        with open(results_path, 'w') as f:
            json.dump([
                {
```

```
                    "test_id": r.test_id,
                    "passed": r.passed,
                    "policy_score": r.policy_score.overall,
                    "format_score": r.format_score.overall,
                    "content_score": r.content_score.overall,
                    "latency_ms": r.latency_ms,
                    "failures": r.failures,
                    "response": r.response[:1000],
                }
                for r in results
            ], f, indent=2)

        print(f"\nEvaluation complete!")
        print(f"  Pass rate: {summary.pass_rate:.1%}")
        print(f"  Critical pass rate: {summary.critical_pass_rate:.1%}")
        print(f"  Report: {report_path}")

        return summary
```

Phase 6: API Integration

> **Purpose**: Configure and integrate OpenAI GPT 5.2 and GPT 5.2 Codex APIs for V3 data augmentation pipeline, including client setup, rate limiting, error handling, and cost optimization.
>
> **Models**: - `gpt-5.2` - General augmentation (conversation rewriting, canonicalization, DPO pairs) - `gpt-5.2-codex` - Agentic coding tasks (repo worm, code generation, diff creation)
>
> **Implementation File**: `rag_plusplus/ml/cognitivetwin_v3/api/openai_client.py`

---

1. OpenAI API Configuration

1.1. API Models

```
from enum import Enum
from dataclasses import dataclass

class OpenAIModel(str, Enum):
    """Available OpenAI models for V3."""

    GPT_5_2 = "gpt-5.2"
    GPT_5_2_CODEX = "gpt-5.2-codex"
    GPT_4_1 = "gpt-4.1"  # Fallback
    GPT_4_TURBO = "gpt-4-turbo"  # Fallback

@dataclass
class ModelConfig:
    """Configuration for a model."""

    model_id: str
    context_length: int
    max_output_tokens: int
    supports_responses_api: bool
    supports_chat_api: bool
    price_per_million_input: float
    price_per_million_output: float

MODEL_CONFIGS = {
    OpenAIModel.GPT_5_2: ModelConfig(
        model_id="gpt-5.2",
        context_length=200000,
        max_output_tokens=16384,
        supports_responses_api=True,
        supports_chat_api=True,
        price_per_million_input=2.50,
        price_per_million_output=10.00,
    ),
    OpenAIModel.GPT_5_2_CODEX: ModelConfig(
        model_id="gpt-5.2-codex",
        context_length=200000,
        max_output_tokens=16384,
        supports_responses_api=True,
        supports_chat_api=False,  # Responses API only
        price_per_million_input=5.00,
        price_per_million_output=15.00,
    ),
}
```

1.2. Client Configuration

```
from dataclasses import dataclass, field
from typing import Optional
import os

@dataclass
class OpenAIClientConfig:
    """Configuration for OpenAI client."""

    # Authentication
    api_key: Optional[str] = None
    organization: Optional[str] = None

    # Default model settings
    default_model: OpenAIModel = OpenAIModel.GPT_5_2
    codex_model: OpenAIModel = OpenAIModel.GPT_5_2_CODEX

    # Request settings
    default_temperature: float = 0.3
    default_max_tokens: int = 4096
    timeout: int = 60

    # Rate limiting
    max_requests_per_minute: int = 500
    max_tokens_per_minute: int = 200000

    # Retry settings
    max_retries: int = 3
    retry_delay: float = 1.0
    exponential_backoff: bool = True

    # Cost tracking
    track_costs: bool = True
    max_cost_per_run: float = 100.0

    def __post_init__(self):
        if self.api_key is None:
            self.api_key = os.environ.get("OPENAI_API_KEY")
        if self.organization is None:
            self.organization = os.environ.get("OPENAI_ORG_ID")
```

1.3. Client Initialization

```
from openai import OpenAI, AsyncOpenAI
import asyncio
from typing import Optional

class V3OpenAIClient:
    """OpenAI client for V3 data augmentation."""

    def __init__(self, config: OpenAIClientConfig = None):
        self.config = config or OpenAIClientConfig()

        # Initialize sync and async clients
        self.client = OpenAI(
            api_key=self.config.api_key,
            organization=self.config.organization,
            timeout=self.config.timeout,
        )

        self.async_client = AsyncOpenAI(
            api_key=self.config.api_key,
            organization=self.config.organization,
            timeout=self.config.timeout,
        )

        # Rate limiter
        self.rate_limiter = RateLimiter(
            max_requests_per_minute=self.config.max_requests_per_minute,
            max_tokens_per_minute=self.config.max_tokens_per_minute,
        )

        # Cost tracker
        self.cost_tracker = CostTracker() if self.config.track_costs else None

    def verify_connection(self) -> bool:
        """Verify API connection."""
        try:
            models = self.client.models.list()
            return len(list(models)) > 0
        except Exception as e:
            print(f"Connection failed: {e}")
            return False
```

---

2. API Endpoints

2.1. Chat Completions API (GPT 5.2)

```
from dataclasses import dataclass
from typing import Optional, List

@dataclass
class ChatMessage:
    role: str
    content: str

@dataclass
class ChatCompletionRequest:
    messages: List[ChatMessage]
    model: str = "gpt-5.2"
    temperature: float = 0.3
    max_tokens: int = 4096
    top_p: float = 1.0
    frequency_penalty: float = 0.0
    presence_penalty: float = 0.0
    stop: Optional[List[str]] = None

class ChatCompletionsAPI:
    """Chat Completions API wrapper."""

    def __init__(self, client: V3OpenAIClient):
        self.client = client

    def complete(
        self,
        messages: List[dict],
        temperature: float = 0.3,
        max_tokens: int = 4096,
    ) -> str:
        """Synchronous chat completion."""

        # Rate limit
        self.client.rate_limiter.wait()

        response = self.client.client.chat.completions.create(
            model=self.client.config.default_model.value,
            messages=messages,
            temperature=temperature,
            max_tokens=max_tokens,
        )

        # Track costs
        if self.client.cost_tracker:
            self.client.cost_tracker.add_usage(
                model=self.client.config.default_model,
                input_tokens=response.usage.prompt_tokens,
                output_tokens=response.usage.completion_tokens,
            )

        return response.choices[0].message.content

    async def complete_async(
        self,
        messages: List[dict],
        temperature: float = 0.3,
        max_tokens: int = 4096,
```

```
    ) -> str:
        """Asynchronous chat completion."""

        await self.client.rate_limiter.wait_async()

        response = await self.client.async_client.chat.completions.create(
            model=self.client.config.default_model.value,
            messages=messages,
            temperature=temperature,
            max_tokens=max_tokens,
        )

        if self.client.cost_tracker:
            self.client.cost_tracker.add_usage(
                model=self.client.config.default_model,
                input_tokens=response.usage.prompt_tokens,
                output_tokens=response.usage.completion_tokens,
            )

        return response.choices[0].message.content

    async def batch_complete(
        self,
        message_batches: List[List[dict]],
        temperature: float = 0.3,
        max_tokens: int = 4096,
        concurrency: int = 5,
    ) -> List[str]:
        """Batch completions with concurrency control."""

        semaphore = asyncio.Semaphore(concurrency)

        async def complete_one(messages):
            async with semaphore:
                return await self.complete_async(messages, temperature, max_tokens)

        tasks = [complete_one(msgs) for msgs in message_batches]
        return await asyncio.gather(*tasks)
```

2.2. Responses API (GPT 5.2 Codex)

```
@dataclass
class ResponsesRequest:
    input: str
    model: str = "gpt-5.2-codex"
    temperature: float = 0.2
    max_tokens: int = 8192

class ResponsesAPI:
    """Responses API wrapper for Codex."""

    def __init__(self, client: V3OpenAIClient):
        self.client = client

    def generate(
        self,
        input_text: str,
        temperature: float = 0.2,
        max_tokens: int = 8192,
    ) -> str:
        """Synchronous code generation."""

        self.client.rate_limiter.wait()

        response = self.client.client.responses.create(
            model=self.client.config.codex_model.value,
            input=input_text,
            temperature=temperature,
            max_tokens=max_tokens,
        )

        if self.client.cost_tracker:
            self.client.cost_tracker.add_usage(
                model=self.client.config.codex_model,
                input_tokens=response.usage.input_tokens,
                output_tokens=response.usage.output_tokens,
            )

        return response.output

    async def generate_async(
        self,
        input_text: str,
        temperature: float = 0.2,
        max_tokens: int = 8192,
    ) -> str:
        """Asynchronous code generation."""

        await self.client.rate_limiter.wait_async()

        response = await self.client.async_client.responses.create(
            model=self.client.config.codex_model.value,
            input=input_text,
            temperature=temperature,
            max_tokens=max_tokens,
        )

        if self.client.cost_tracker:
            self.client.cost_tracker.add_usage(
```

```
                model=self.client.config.codex_model,
                input_tokens=response.usage.input_tokens,
                output_tokens=response.usage.output_tokens,
            )

        return response.output
```

---

3. Rate Limiting

3.1. Token Bucket Rate Limiter

```
import time
import asyncio
from threading import Lock

class RateLimiter:
    """Token bucket rate limiter for API calls."""

    def __init__(
        self,
        max_requests_per_minute: int = 500,
        max_tokens_per_minute: int = 200000,
    ):
        self.max_requests = max_requests_per_minute
        self.max_tokens = max_tokens_per_minute

        # Token buckets
        self.request_tokens = max_requests_per_minute
        self.token_tokens = max_tokens_per_minute

        # Timing
        self.last_refill = time.time()
        self.refill_interval = 60.0  # 1 minute

        # Synchronization
        self.lock = Lock()
        self.async_lock = asyncio.Lock()

    def _refill(self):
        """Refill token buckets based on elapsed time."""
        now = time.time()
        elapsed = now - self.last_refill

        if elapsed >= self.refill_interval:
            self.request_tokens = self.max_requests
            self.token_tokens = self.max_tokens
            self.last_refill = now
        else:
            # Partial refill
            fraction = elapsed / self.refill_interval
            self.request_tokens = min(
                self.max_requests,
                self.request_tokens + int(self.max_requests * fraction)
            )
            self.token_tokens = min(
                self.max_tokens,
                self.token_tokens + int(self.max_tokens * fraction)
            )

    def wait(self, estimated_tokens: int = 1000):
        """Wait for rate limit, blocking."""

        with self.lock:
            self._refill()

            while self.request_tokens <= 0 or self.token_tokens < estimated_tokens:
                sleep_time = 0.1
                time.sleep(sleep_time)
                self._refill()
```

```
            self.request_tokens -= 1
            self.token_tokens -= estimated_tokens

    async def wait_async(self, estimated_tokens: int = 1000):
        """Wait for rate limit, async."""

        async with self.async_lock:
            self._refill()

            while self.request_tokens <= 0 or self.token_tokens < estimated_tokens:
                await asyncio.sleep(0.1)
                self._refill()

            self.request_tokens -= 1
            self.token_tokens -= estimated_tokens
```

3.2. Adaptive Rate Limiting

```
class AdaptiveRateLimiter(RateLimiter):
    """Rate limiter that adapts to API responses."""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.consecutive_rate_limits = 0
        self.backoff_multiplier = 1.0

    def on_success(self):
        """Call on successful request."""
        self.consecutive_rate_limits = 0
        self.backoff_multiplier = max(1.0, self.backoff_multiplier * 0.9)

    def on_rate_limit(self):
        """Call on rate limit error."""
        self.consecutive_rate_limits += 1
        self.backoff_multiplier = min(10.0, self.backoff_multiplier * 2.0)

    def get_wait_time(self) -> float:
        """Get wait time with backoff."""
        base_wait = 60.0 / self.max_requests
        return base_wait * self.backoff_multiplier
```

---

4. Error Handling

4.1. Error Types

```
from enum import Enum
from dataclasses import dataclass
from typing import Optional

class APIErrorType(str, Enum):
    RATE_LIMIT = "rate_limit"
    AUTHENTICATION = "authentication"
    INVALID_REQUEST = "invalid_request"
    SERVER_ERROR = "server_error"
    TIMEOUT = "timeout"
    CONTENT_FILTER = "content_filter"
    CONTEXT_LENGTH = "context_length"
    UNKNOWN = "unknown"

@dataclass
class APIError:
    error_type: APIErrorType
    message: str
    status_code: Optional[int] = None
    retry_after: Optional[float] = None
```

4.2. Error Handler

```
from openai import (
    RateLimitError,
    AuthenticationError,
    BadRequestError,
    APIStatusError,
    APITimeoutError,
)

class ErrorHandler:
    """Handle API errors with retry logic."""

    def __init__(self, config: OpenAIClientConfig):
        self.config = config

    def classify_error(self, error: Exception) -> APIError:
        """Classify an exception into APIError."""

        if isinstance(error, RateLimitError):
            retry_after = self._extract_retry_after(error)
            return APIError(
                error_type=APIErrorType.RATE_LIMIT,
                message=str(error),
                status_code=429,
                retry_after=retry_after,
            )

        elif isinstance(error, AuthenticationError):
            return APIError(
                error_type=APIErrorType.AUTHENTICATION,
                message=str(error),
                status_code=401,
            )

        elif isinstance(error, BadRequestError):
            if "context_length" in str(error).lower():
                return APIError(
                    error_type=APIErrorType.CONTEXT_LENGTH,
                    message=str(error),
                    status_code=400,
                )
            elif "content_filter" in str(error).lower():
                return APIError(
                    error_type=APIErrorType.CONTENT_FILTER,
                    message=str(error),
                    status_code=400,
                )
            return APIError(
                error_type=APIErrorType.INVALID_REQUEST,
                message=str(error),
                status_code=400,
            )

        elif isinstance(error, APIStatusError):
            return APIError(
                error_type=APIErrorType.SERVER_ERROR,
                message=str(error),
                status_code=getattr(error, 'status_code', 500),
            )
```

```
        elif isinstance(error, APITimeoutError):
            return APIError(
                error_type=APIErrorType.TIMEOUT,
                message=str(error),
            )

        return APIError(
            error_type=APIErrorType.UNKNOWN,
            message=str(error),
        )

    def should_retry(self, error: APIError, attempt: int) -> bool:
        """Determine if error is retryable."""

        if attempt >= self.config.max_retries:
            return False

        retryable_types = {
            APIErrorType.RATE_LIMIT,
            APIErrorType.SERVER_ERROR,
            APIErrorType.TIMEOUT,
        }

        return error.error_type in retryable_types

    def get_retry_delay(self, error: APIError, attempt: int) -> float:
        """Calculate retry delay."""

        if error.retry_after:
            return error.retry_after

        base_delay = self.config.retry_delay

        if self.config.exponential_backoff:
            return base_delay * (2 ** attempt)

        return base_delay

    def _extract_retry_after(self, error: RateLimitError) -> Optional[float]:
        """Extract retry-after from rate limit error."""

        import re

        message = str(error)
        match = re.search(r"try again in (\d+(?:\.\d+)?)\s*s", message)

        if match:
            return float(match.group(1))

        return None
```

4.3. Retry Decorator

```
import functools
import asyncio

def with_retry(func):
    """Decorator to add retry logic to API calls."""

    @functools.wraps(func)
    async def wrapper(self, *args, **kwargs):
        handler = ErrorHandler(self.client.config)

        for attempt in range(self.client.config.max_retries + 1):
            try:
                result = await func(self, *args, **kwargs)

                if hasattr(self.client, 'rate_limiter'):
                    if hasattr(self.client.rate_limiter, 'on_success'):
                        self.client.rate_limiter.on_success()

                return result

            except Exception as e:
                error = handler.classify_error(e)

                if hasattr(self.client, 'rate_limiter'):
                    if hasattr(self.client.rate_limiter, 'on_rate_limit'):
                        if error.error_type == APIErrorType.RATE_LIMIT:
                            self.client.rate_limiter.on_rate_limit()

                if not handler.should_retry(error, attempt):
                    raise

                delay = handler.get_retry_delay(error, attempt)
                print(f"Retry {attempt + 1} after {delay:.1f}s: {error.message[:100]}")
                await asyncio.sleep(delay)

        raise Exception("Max retries exceeded")

    return wrapper
```

---

5. Cost Tracking

5.1. Cost Tracker

```
from dataclasses import dataclass, field
from datetime import datetime
from threading import Lock

@dataclass
class UsageRecord:
    timestamp: datetime
    model: OpenAIModel
    input_tokens: int
    output_tokens: int
    cost: float

class CostTracker:
    """Track API usage and costs."""

    def __init__(self):
        self.records: List[UsageRecord] = []
        self.total_cost = 0.0
        self.lock = Lock()

    def add_usage(
        self,
        model: OpenAIModel,
        input_tokens: int,
        output_tokens: int,
    ):
        """Record usage."""

        config = MODEL_CONFIGS[model]

        input_cost = (input_tokens / 1_000_000) * config.price_per_million_input
        output_cost = (output_tokens / 1_000_000) * config.price_per_million_output
        cost = input_cost + output_cost

        record = UsageRecord(
            timestamp=datetime.utcnow(),
            model=model,
            input_tokens=input_tokens,
            output_tokens=output_tokens,
            cost=cost,
        )

        with self.lock:
            self.records.append(record)
            self.total_cost += cost

    def get_summary(self) -> dict:
        """Get usage summary."""

        by_model = {}

        for record in self.records:
            model_key = record.model.value
            if model_key not in by_model:
                by_model[model_key] = {
                    "requests": 0,
                    "input_tokens": 0,
                    "output_tokens": 0,
```

```
                    "cost": 0.0,
                }

            by_model[model_key]["requests"] += 1
            by_model[model_key]["input_tokens"] += record.input_tokens
            by_model[model_key]["output_tokens"] += record.output_tokens
            by_model[model_key]["cost"] += record.cost

        return {
            "total_requests": len(self.records),
            "total_cost": self.total_cost,
            "by_model": by_model,
        }

    def check_budget(self, max_cost: float) -> bool:
        """Check if within budget."""
        return self.total_cost <= max_cost
```

5.2. Cost Guard

```
class CostGuard:
    """Guard against excessive API costs."""

    def __init__(
        self,
        tracker: CostTracker,
        max_cost: float = 100.0,
        warn_at: float = 0.8,
    ):
        self.tracker = tracker
        self.max_cost = max_cost
        self.warn_at = warn_at
        self.warned = False

    def check(self) -> bool:
        """Check if within budget. Returns False if over budget."""

        current = self.tracker.total_cost

        if current >= self.max_cost:
            raise Exception(f"Cost limit exceeded: ${current:.2f} >= ${self.max_cost:.2f}")

        if current >= self.max_cost * self.warn_at and not self.warned:
            print(f"WARNING: Cost approaching limit: ${current:.2f} / ${self.max_cost:.2f}")
            self.warned = True

        return True
```

---

6. Prompt Templates

6.1. Corpus Surgery Templates

```
REWRITER_SYSTEM_PROMPT = """You are a response rewriter for CognitiveTwin V3.

Your task is to rewrite assistant responses that asked for unnecessary permission
into responses that execute immediately.

RULES:
1. NEVER ask questions when the directive is clear
2. NEVER use phrases like "Would you like me to...", "Should I...", "Can you confirm..."
3. If assumptions are needed, STATE them as declarations, then proceed
4. ALWAYS produce the artifact that was requested
5. Keep the same technical content, just remove permission-seeking

ASSUMPTION PROTOCOL:
- If something is unknown but non-blocking: choose a reasonable default
- State assumptions in ONE line at the start: "Assumptions: ..."
- Then proceed with the implementation/answer
- NO question marks after assumptions

OUTPUT FORMAT:
Return ONLY the rewritten assistant response. No explanations.
"""

CLASSIFIER_PROMPT = """Classify the following assistant response as one of:
- UNJUSTIFIED: Asks permission when not needed (directive was clear)
- JUSTIFIED: Asks for genuinely missing information
- NEUTRAL: Neither permission-seeking nor requiring clarification

User message: {user_message}
Assistant response: {assistant_response}
Directive completeness: {directive_completeness}

Classification:"""
```

6.2. Repo Worm Templates

```
CODEX_SYSTEM_PROMPT = """You are a code generation assistant for CognitiveTwin V3.

RULES:
1. Generate complete, working code that compiles
2. Follow existing patterns and conventions in the codebase
3. Do NOT ask clarifying questions - make reasonable assumptions
4. State assumptions briefly at the start if needed
5. Include proper error handling and edge cases
6. Match the existing code style exactly

OUTPUT FORMAT:
- Provide code in markdown code blocks
- Include brief comments explaining complex logic
- For diffs, use unified diff format

ASSUMPTION PROTOCOL:
If you need to assume something:
- State it in a comment: # Assumption: ...
- Then proceed with implementation
- NO questions
"""

TASK_PROMPT_TEMPLATE = """
REPOSITORY CONTEXT:
{context}

TASK:
{task_description}

CONSTRAINTS:
- Must compile without errors
- Follow existing patterns
- No new dependencies unless necessary
- Do NOT ask questions

Provide the implementation:
"""
```

6.3. Conversation Worm Templates

```
PARAPHRASE_PROMPT = """Generate {count} paraphrases of the user message that:
1. Preserve the exact same intent and meaning
2. Use different wording, sentence structure, or phrasing
3. Maintain the same level of formality
4. Keep any technical terms unchanged

Original message:
{message}

Output format:
PARAPHRASE 1: ...
PARAPHRASE 2: ...
"""

IDEAL_RESPONSE_PROMPT = """Generate the ideal assistant response for CognitiveTwin V3.

The original assistant response asked for permission or clarification when it shouldn't have.
Generate what the assistant SHOULD have said instead.

RULES:
1. Execute immediately - do not ask permission
2. If assumptions are needed, state them briefly then proceed
3. Produce the requested artifact/output
4. Do NOT end with a question
5. Match the technical level and style of the conversation

CONVERSATION CONTEXT:
{context}

USER MESSAGE:
{user_message}

PROBLEMATIC RESPONSE:
{problematic_response}

Generate the ideal response:
"""
```

---

7. V3 Augmentation Pipeline

7.1. Unified Client

```
class V3AugmentationClient:
    """Unified client for V3 data augmentation."""

    def __init__(self, config: OpenAIClientConfig = None):
        self.client = V3OpenAIClient(config)
        self.chat_api = ChatCompletionsAPI(self.client)
        self.responses_api = ResponsesAPI(self.client)
        self.cost_guard = CostGuard(
            self.client.cost_tracker,
            max_cost=config.max_cost_per_run if config else 100.0,
        )

    async def rewrite_assistant_turn(
        self,
        assistant_message: str,
        user_message: str,
        conversation_history: List[dict],
    ) -> str:
        """Rewrite an assistant turn using GPT 5.2."""

        self.cost_guard.check()

        messages = [
            {"role": "system", "content": REWRITER_SYSTEM_PROMPT},
            {"role": "user", "content": f"""CONVERSATION HISTORY:
{self._format_history(conversation_history)}

USER MESSAGE:
{user_message}

PROBLEMATIC ASSISTANT RESPONSE:
{assistant_message}

Rewrite this response to execute immediately:"""}
        ]

        return await self.chat_api.complete_async(
            messages,
            temperature=0.3,
            max_tokens=4096,
        )

    async def generate_code(
        self,
        task_description: str,
        context: str,
    ) -> str:
        """Generate code using GPT 5.2 Codex."""

        self.cost_guard.check()

        prompt = TASK_PROMPT_TEMPLATE.format(
            context=context,
            task_description=task_description,
        )

        full_input = f"{CODEX_SYSTEM_PROMPT}\n\n{prompt}"
```

```
        return await self.responses_api.generate_async(
            full_input,
            temperature=0.2,
            max_tokens=8192,
        )

    async def generate_paraphrases(
        self,
        message: str,
        count: int = 2,
    ) -> List[str]:
        """Generate paraphrases of a message."""

        self.cost_guard.check()

        messages = [
            {"role": "system", "content": "You are a paraphrase generator."},
            {"role": "user", "content": PARAPHRASE_PROMPT.format(
                count=count,
                message=message,
            )}
        ]

        response = await self.chat_api.complete_async(
            messages,
            temperature=0.7,
            max_tokens=1024,
        )

        # Parse paraphrases
        import re
        paraphrases = []
        for match in re.finditer(r'PARAPHRASE \d+:\s*(.+?)(?=PARAPHRASE \d+:|$)', response, re.DOTA
            paraphrases.append(match.group(1).strip())

        return paraphrases[:count]

    async def generate_ideal_response(
        self,
        user_message: str,
        problematic_response: str,
        context: List[dict],
    ) -> str:
        """Generate ideal response for friction point."""

        self.cost_guard.check()

        messages = [
            {"role": "system", "content": IDEAL_RESPONSE_PROMPT.format(
                context=self._format_history(context),
                user_message=user_message,
                problematic_response=problematic_response,
            )}
        ]

        return await self.chat_api.complete_async(
            messages,
            temperature=0.3,
            max_tokens=4096,
```

```
        )

    def get_cost_summary(self) -> dict:
        """Get cost summary."""
        return self.client.cost_tracker.get_summary() if self.client.cost_tracker else {}

    def _format_history(self, history: List[dict]) -> str:
        """Format conversation history."""
        parts = []
        for msg in history[-6:]:
            role = msg.get("role", "unknown").capitalize()
            content = msg.get("content", "")[:500]
            parts.append(f"{role}: {content}")
        return "\n\n".join(parts)
```

7.2. Batch Processing

```
class BatchProcessor:
    """Process data in batches with progress tracking."""

    def __init__(
        self,
        client: V3AugmentationClient,
        batch_size: int = 10,
        concurrency: int = 5,
    ):
        self.client = client
        self.batch_size = batch_size
        self.concurrency = concurrency

    async def process_rewrites(
        self,
        items: List[dict],
        progress_callback = None,
    ) -> List[dict]:
        """Process rewrite tasks in batches."""

        results = []

        for i in range(0, len(items), self.batch_size):
            batch = items[i:i + self.batch_size]

            tasks = [
                self.client.rewrite_assistant_turn(
                    item["assistant_message"],
                    item["user_message"],
                    item.get("history", []),
                )
                for item in batch
            ]

            batch_results = await asyncio.gather(*tasks, return_exceptions=True)

            for item, result in zip(batch, batch_results):
                if isinstance(result, Exception):
                    results.append({**item, "error": str(result)})
                else:
                    results.append({**item, "rewritten": result})

            if progress_callback:
                progress_callback(min(i + self.batch_size, len(items)), len(items))

        return results

    async def process_code_generation(
        self,
        tasks: List[dict],
        progress_callback = None,
    ) -> List[dict]:
        """Process code generation tasks in batches."""

        results = []

        for i in range(0, len(tasks), self.batch_size):
            batch = tasks[i:i + self.batch_size]
```

```
            gen_tasks = [
                self.client.generate_code(
                    task["description"],
                    task["context"],
                )
                for task in batch
            ]

            batch_results = await asyncio.gather(*gen_tasks, return_exceptions=True)

            for task, result in zip(batch, batch_results):
                if isinstance(result, Exception):
                    results.append({**task, "error": str(result)})
                else:
                    results.append({**task, "generated": result})

            if progress_callback:
                progress_callback(min(i + self.batch_size, len(tasks)), len(tasks))

        return results
```

---

8. CLI Interface

```
import click
import asyncio
from pathlib import Path

@click.group()
def cli():
    """OpenAI API tools for CognitiveTwin V3."""
    pass

@cli.command()
def verify():
    """Verify API connection."""

    client = V3OpenAIClient()
    if client.verify_connection():
        click.echo("✓ API connection verified")
    else:
        click.echo("✗ API connection failed")

@cli.command()
@click.option("--input", type=Path, required=True, help="Input JSONL file")
@click.option("--output", type=Path, required=True, help="Output JSONL file")
@click.option("--batch-size", default=10, help="Batch size")
def rewrite(input, output, batch_size):
    """Rewrite assistant turns."""

    import json

    # Load data
    with open(input) as f:
        items = [json.loads(line) for line in f]

    client = V3AugmentationClient()
    processor = BatchProcessor(client, batch_size=batch_size)

    def progress(current, total):
        click.echo(f"  Progress: {current}/{total}")

    results = asyncio.run(processor.process_rewrites(items, progress))

    # Save results
    with open(output, 'w') as f:
        for result in results:
            f.write(json.dumps(result) + '\n')

    # Print summary
    summary = client.get_cost_summary()
    click.echo(f"\nComplete!")
    click.echo(f"  Total cost: ${summary.get('total_cost', 0):.2f}")

@cli.command()
def cost_report():
    """Show cost report."""

    client = V3AugmentationClient()
    summary = client.get_cost_summary()

    click.echo("Cost Summary:")
```

```
    click.echo(f"  Total requests: {summary.get('total_requests', 0)}")
    click.echo(f"  Total cost: ${summary.get('total_cost', 0):.2f}")

    for model, data in summary.get('by_model', {}).items():
        click.echo(f"\n  {model}:")
        click.echo(f"    Requests: {data['requests']}")
        click.echo(f"    Input tokens: {data['input_tokens']:,}")
        click.echo(f"    Output tokens: {data['output_tokens']:,}")
        click.echo(f"    Cost: ${data['cost']:.2f}")

if __name__ == "__main__":
    cli()
```

---

9. Environment Setup

9.1. Required Environment Variables

```
# Required
export OPENAI_API_KEY="sk-..."

# Optional
export OPENAI_ORG_ID="org-..."
export OPENAI_MAX_REQUESTS_PER_MINUTE=500
export OPENAI_MAX_TOKENS_PER_MINUTE=200000
export OPENAI_MAX_COST_PER_RUN=100.0
```

9.2. Dependencies

```
# requirements.txt
openai>=1.40.0
asyncio
aiohttp
tenacity
```

9.3. Configuration File

```
# config/openai.yaml
api:
  default_model: gpt-5.2
  codex_model: gpt-5.2-codex
  timeout: 60

rate_limits:
  max_requests_per_minute: 500
  max_tokens_per_minute: 200000

retry:
  max_retries: 3
  retry_delay: 1.0
  exponential_backoff: true

costs:
  track_costs: true
  max_cost_per_run: 100.0
  warn_at_percentage: 80
```
