ChatGPT

Claude 자가개선 루프 시스템 코드 레벨 개선 검토 보고서

본 보고서는 Claude 기반 자가개선 루프 시스템에 대한 코드 수준의 개선 계획을 검토한 것입니다. 사용자가 제공한 **CODE-IMPROVEMENT-PLAN.md** 문서와 그 보완 설명(연구 보고서)을 바탕으로, 계획된 개선 사항들이 실제 코드 구조(여러 에이전트 모듈)와 어떻게 연계되는지, 누락되거나 위험한 요소는 무엇이며 추가 리팩토링이 필요한 부분은 어디인지를 진단합니다. 각 섹션에서는 개선 항목별 구현 변경 사항, 고려해야 할 점(충돌 지점, 위험성 등), 그리고 코드 예시나 실행 계획을 제시합니다.

1. 개선 항목별 코드 수정 및 잠재 충돌 분석

이 섹션에서는 식별된 6개의 핵심 개선 사항을 하나씩 다루면서, 해당 기능 개선이 구체적으로 어떤 모듈의 코드 변경을 요구하는지와 예상되는 충돌 또는 부작용을 분석합니다. 다루는 모듈은 `relationship_definer`, `meta_orchestrator`, `self_improver_agent`, `socratic_mediator_agent`, `quality_agent` (신규), `relationship_ontology` (신규) 등입니다.

개선 1: RelationshipDefiner – 다차원 불확실성 모델링

- **코드 변경 사항:** 관계 정의 에이전트(`agents/relationship_definer.py`)의 **JSON 출력 스키마**를 확장하고 **프롬프트**를 강화합니다. 기존에는 각 관계 판별 결과로 하나의 `confidence` 점수만 제공하여 모든 불확실성을 단일 수치로 표현했습니다. 이를 개선하기 위해 JSON에 세 가지 불확실성 요소(`epistemic`, `aleatoric`, `model_indecision`)의 breakdown과 원인 코드(`uncertainty_reason`), 대안 분류안 목록(`alternative_classifications`)을 추가합니다 [1]. 예를 들면 다음과 같이 출력 구조가 바뀝니다:

```
json
  {
    "confidence": 0.98,
    "uncertainty_breakdown": {
      "epistemic": 0.05,
      "aleatoric": 0.10,
      "model_indecision": 0.02
    },
    "uncertainty_reason": "concept_B_context_insufficient",
    "alternative_classifications": [
      {"type": "co-requisite", "confidence": 0.65}
    ]
  }
```
[1]

또한 **시스템 프롬프트**에 "## UNCERTAINTY ANALYSIS" 섹션을 추가하여, 각 관계에 대해 **세 가지 종류의 불확실성**을 평가하도록 LLM에 지시합니다 [2] [3]. 이 프롬프트 조각은 Epistemic(지식 부족), Aleatoric(내재적 모호성), Model Indecision(모델 판정 불안정)의 정의와 스코어 범위를 설명하고, 불확실성 이유를 몇 가지 코드 형태로 표기하도록 합니다. 예를 들어 `"concept_X_context_insufficient"`나 `"boundary_ambiguous_between_TYPE1_TYPE2"` 등의 원인 코드를 출력하게 지시하고, **신뢰도 0.85 미만**인

경우 대안 관계 분류안을 함께 제시하도록 명시합니다 [2] [4]. (연구 보고서에서도 "신뢰도 0.70 이상만 출력" 규칙을 재확인하여 정확도를 유지하라고 권고하고 있습니다 [5].)

- **연계 및 충돌 요소:** 출력 스키마에 새로운 필드들이 추가됨에 따라, **이 출력을 이용하는 다른 모듈들의 파싱 로직**을 점검해야 합니다. 다행히 새로운 필드는 기존 구조에 **추가적인 키**로 붙는 **Additive 변경**이므로, 만약 다른 컴포넌트가 해당 필드를 참조하지 않더라도 큰 오류는 발생하지 않습니다 (백워드 호환성 유지) [6]. 그러나 이상적으로는 **SocraticMediator**나 **SelfImproverAgent** 등이 RelationshipDefiner의 결과를 활용할 때, 새로운 `uncertainty_reason`나 `alternative_classifications` 정보를 인지하고 활용하도록 업데이트해야 합니다. 예를 들어 SocraticMediator 에이전트는 `uncertainty_reason`을 읽어 해당 이유에 맞춰 **후속 질문**을 던지는 진단 루프를 구현할 수 있습니다 [7]. 실제 개선 계획에서도 `"concept_B_context_insufficient"` 같은 이유 코드를 파악하여 Self-Improver가 추가 컨텍스트를 제공하는 식으로 활용될 수 있다고 명시하고 있습니다 [7]. 이러한 **피드백 루프**를 위해 SocraticMediator의 프롬프트나 로직을 조정해야 할 것입니다 (예: 불확실성 원인에 따라 "개념 B에 대한 배경을 좀 더 설명해주세요" 같은 질문을 자동 생성).

- **추가 고려사항:** 불확실성 세분화에 따른 **데이터 모델 변경**도 필요합니다. 만약 RelationshipDefiner 결과를 담는 Python 객체나 데이터 클래스가 있다면, 여기에 `uncertainty_breakdown` 등의 필드를 추가하고 기본값을 처리해야 합니다. 또한 새로운 정보들은 **UI나 출력 보고서에 어떻게 노출할지** 결정해야 합니다. Confidence가 높아도 Epistemic/Aleatoric 불확실성이 큰 경우 사용자에게 표기할지, 또는 `alternative_classifications` 목록은 최종 결과에 포함할지 등에 대한 정책이 필요합니다 (예: 불확실성이 높으면 UI에 경고 아이콘 표시 등). 마지막으로, LLM 프롬프트가 길어지므로 **토큰 비용**이 약간 증가할 수 있지만, 개선된 정보로 **반복 진단 횟수를 줄여** 결과적으로 효율이 좋아질 것으로 기대됩니다 [7].

개선 2: RelationshipDefiner – 연쇄적 사고(Chain-of-Thought) 프롬프트 도입

- **코드 변경 사항:** RelationshipDefiner의 시스템 프롬프트에 **Chain-of-Thought (CoT)** 방식을 도입하여, **응답에 추론 과정을 함께 포함**하도록 개선합니다. 구체적으로 프롬프트에 **2단계 출력 구조**를 명시합니다. **Phase 1: ANALYSIS** 단계에서는 모델이 중간 **추론 과정**을 텍스트로 작성하고, **Phase 2: CLASSIFICATION** 단계에서 최종 JSON 출력만을 제공하도록 유도합니다 [8] [9]. Phase 1에서는 다음과 같은 논리적 단계를 거치도록 템플릿을 제시합니다:

- **Step 1 - 개념 분해**: 개념 A, B의 핵심 정의를 나열하고 관련 수학 구조(연산, 객체, 성질)를 식별한다 [10].

- **Step 2 - 의존성 조사**: "B를 이해하는데 A가 필요한가?", "B의 정의에 A가 언급되는가?" 등을 yes/no와 증거와 함께 판단하고, 함의된 전제 조건들을 정리한다 [11].
- **Step 3 - 관계 가설 설정**: 후보 관계 유형 2~3가지를 열거하고, 각 후보에 대해 근거와 반증을 대조한다. 그 중 **유력한 후보**를 선정한다 [12].
- **Step 4 - 검증**: 후보 관계에 대해 **논리 검증**을 시행한다. 예를 들어 **정체성(동일 개념 여부 확인)**, **추이성 테스트** (A→B, B→C 존재 시 A→C 성립 여부), **방향성 확인** (관계가 방향성을 가지는지), **순환 여부** (이 관계로 인해 A→...→A 형태의 사이클이 생기지 않는지) 등을 점검하도록 한다 [13]. (이 내용은 연구 보고서에서 언급된 OntoClean 검증 규칙 – 정체성, 추이성, 순환 금지 등을 프롬프트에 포함하라는 제언과 궤를 같이합니다 [5].)

이후 **Phase 2: CLASSIFICATION** 단계에서는 위 분석을 바탕으로 오직 JSON 형식의 최종 출력만 생성하게 합니다 [14]. 이 프롬프트 구조를 삽입함으로써, LLM이 **사고 과정을 먼저 글로 설명**한 뒤 결과를 내놓도록 유도합니다.

- **응답 파싱 로직:** 이러한 2단계 출력으로 인해 LLM의 응답은 사람 가독형 텍스트 + JSON이 결합된 형태가 되므로, 이를 처리하기 위해 **파싱 함수를 업데이트**해야 합니다. 구체적으로, 응답 문자열에서 Phase 1과 Phase 2 부분을 분리하여 JSON만 추출하고, 추론 과정을 별도로 저장해야 합니다. 예를 들어 아래와 같은 파이썬 코드로 구현할 수 있습니다:

```python # Phase 1 추론 과정 추출 reasoning_trace = "" if "### Phase 1: ANALYSIS" in response_text: phase1_start = response_text.find("### Phase 1: ANALYSIS") phase1_end = response_text.find("### Phase 2: CLASSIFICATION") if phase1_end > phase1_start: reasoning_trace = response_text[phase1_start:phase1_end].strip()

# Phase 2의 JSON 결과 추출 json_text = "" if "json" in response_text: json_start = response_text.find("json") + 7 json_end = response_text.find("```", json_start) json_text = response_text[json_start:json_end].strip()

relationships = json.loads(json_text) # 각 관계 결과에 추론 과정을 첨부 for rel in relationships: rel['reasoning_trace'] = reasoning_trace ``` [15] [16]

위 코드에서는 응답 문자열 내에 `"### Phase 1: ANALYSIS"`와 `"### Phase 2: CLASSIFICATION"` 구분자가 존재할 경우 이를 찾아 **reasoning_trace**로 저장하고, `json` 코드 블록 부분만 떼어내 JSON으로 로드한 뒤 각 관계 딕셔너리에 `reasoning_trace` 필드를 추가하고 있습니다. 기존에 JSON만 오던 출력 구조에 비해 파싱이 복잡해지므로, 이러한 절차를 RelationshipDefiner 모듈의 응답 처리 루틴에 추가해야 합니다. (CoT 적용 이전의 단순 JSON 응답도 여전히 처리할 수 있도록 `elif "```"`로 기존 로직을 보존/분기하는 등 **후방 호환 처리**도 함께 구현합니다 [17].)

- **연계 및 충돌 요소:** 이 개선으로 **LLM 호출 1회당 토큰 수가 증가**하고 응답 길이가 길어지지만, 대신 모델의 **의사결정 근거를 투명하게 파악**할 수 있게 됩니다. SocraticMediator 등 디버깅을 담당하는 에이전트는 이제 각 관계에 대한 추론 흔적(`reasoning_trace`)을 읽어, LLM이 왜 그런 관계를 분류했는지 검증하거나 잘못된 추론이 있었다면 피드백을 줄 수 있습니다 [18]. 이는 **디버깅 용이성**을 크게 높이며, Chain-of-Thought 연구에서 보고된 대로 **복잡한 분류 문제의 정확도 향상(+15~25%)**도 기대할 수 있습니다 [18].

잠재적인 충돌로는, 모델이 프롬프트를 제대로 따르지 않고 형식을 어길 경우 파싱 오류가 발생할 수 있다는 점입니다. 다행히 현재 설계에서는 Phase 1 부분이 없더라도 기존처럼 JSON만 읽으면 되므로 **문제 상황별 폴백(fallback)**이 가능합니다 (계획 문서에서도 이 변경의 **Risk**를 "Low"로 평가했습니다 [19]). 그래도 CoT 프롬프트의 효과를 극대화하려면, **프롬프트 예시**를 충분히 제공하고 형식을 반복 검증해야 합니다. 또 하나 고려할 점은 reasoning_trace의 **저장 위치와 활용 범위**입니다. 이를 단순 로그로만 둘지, 아니면 관계 객체의 속성으로 영구 저장하여 향후 개선 반복에서 참고할지 결정해야 합니다. 후자를 택하면 데이터베이스 스키마나 Relationship 객체에 `reasoning_trace` 필드를 추가로 정의해야 할 것입니다.

## 개선 3: MetaOrchestrator – 동적 품질 게이트(Dynamic Quality Gate) 도입

- **문제 진단:** Self-Improvement 루프에서 **품질 게이트(Quality Gate)**는 변경된 지식 그래프나 코드 개선안을 적용하기 전에 특정 품질 지표(영향 범위, 테스트 커버리지 등)에 따라 통과 여부를 결정하는 역할을 합니다. 현재 구현은 MetaOrchestrator 내에 이 로직이 하드코딩되어 있는데, 예를 들어 **CIS**(Change Impact Size, 영향받은 파일 수)가 20개 이상이면 실패, 테스트 커버리지가 80% 미만이면 실패와 같이 고정 기준이 사용됩니다 [20]. 이러한 **정적 임계값** 설정은 맥락을 반영하지 못해, “변경 파일 19개이면 통과, 20개이면 실패” 같은 경계선 문제나, 반대로 단 1파일 변경이어도 그 파일이 핵심 시스템이면 간과되는 문제를 초래합니다 [20] [21].

- **코드 변경 사항:** 개선안은 **품질 게이트 임계값을 동적으로 계산**하도록 MetaOrchestrator를 보강하는 것입니다. 이를 위해 새로운 설정 모듈 `agents/criticality_config.py`를 만들고, 파일별 **중요도 점수(criticality score)**를 정의합니다. 예컨대 핵심 오케스트레이션 파일(`meta_orchestrator.py` 등)은 10점, 주요 에이전트(`relationship_definer.py` 등)는 8~9점, 테스트나 문서 파일은 1~2점 등으로 분류합니다 [22] [23]. 이 점수표에 기반하여, 품질 게이트 시 **변경 영향 정보(ImpactAnalysis)**에 담긴 파일 목록의 평균/최대 중요도를 계산하고, 그에 따라 허용 파일수(CIS) 임계값과 최소 테스트 커버리지 임계값을 **동적으로 산출**합니다. 예를 들어 아래와 같은 함수로 임계값을 결정할 수 있습니다:
```

```
avg_criticality = sum(scores) / len(scores)
max_criticality = max(scores)
# 중요도가 높을수록 더 엄격한 기준 적용
cis_threshold = max(5, 30 - (avg_criticality * 1.5))
coverage_threshold = min(0.95, 0.65 + (avg_criticality * 0.03))
# 최고 중요도가 9 이상이면 검증 라운드 2회 요구 등 추가 조치
verification_rounds = 2 if max_criticality >= 9 else 1
```【6†L313-L321】【6†L319-L327】

위 로직은 예를 들어 평균 중요도가 높으면 **허용 CIS 크기를 작게**(기본 20에서 감소) 하고 **커버리지 요구치를 높게**(기본 80%에서 증가) 설정합니다. 반대로 변경 파일들이 문서나 예제에 국한되어 중요도가 낮다면 CIS 기준을 늘려주고 커버리지 요구를 다소 낮출 수 있습니다【7†L433-L441】. 실제 계획서에서도 *"Critical files: CIS < 15, Coverage > 0.83 / Documentation: CIS < 25, Coverage > 0.70"*와 같은 맥락별 임계값 조정 예시를 제시하고 있습니다【7†L433-L441】.

구현 면에서, MetaOrchestrator 내 `evaluate_quality_gate` 메서드를 수정하여 **위 계산 함수를 호출**하고, 반환된 임계값을 활용하도록 합니다【6†L353-L360】. Pseudo-code로 하면 다음과 같습니다:

```python
thresholds = calculate_dynamic_thresholds(impact_analysis.affected_files)
cis_thr = thresholds['cis_threshold']
cov_thr = thresholds['coverage_threshold']
if impact_analysis.cis_size >= cis_thr:
    failures.append(f"CIS size {impact_analysis.cis_size} exceeds {cis_thr:.1f} for criticality {thresholds['avg_criticality']:.1f}/10")
if impact_analysis.test_coverage < cov_thr:
    failures.append(f"Test coverage {impact_analysis.test_coverage:.0%} below {cov_thr:.0%} for criticality {thresholds['avg_criticality']:.1f}/10")
# ... (이하 경고 및 결과 구성)
```

여기에 **경고(Warnings)** 로직도 강화하여, 변경에 **매우 중요한 파일(최대 중요도 9~10)이 포함된 경우** 개발자에게 경각심을 주도록 합니다. 예를 들어 "⚠ Mission-critical components affected... 시스템이 자동으로 2회 검증을 실시함" 같은 경고를 출력하고, 실제로 `verification_rounds` 값이 2라면 품질 게이트를 **한 번 더 반복 수행**하거나 추가적인 검증 절차(예: QualityAgent 재실행 또는 HITL 체크포인트)를 거치게 할 수 있습니다. 이러한 경고/재시도 정보를 QualityGate 결과 객체에 포함시켜 상위 로직에서 활용하게 합니다 [25] [26].

- **연계 및 충돌 요소:** 이 개선은 MetaOrchestrator와 새로운 설정 모듈 간의 **의존성**을 추가합니다. `criticality_config.py`를 MetaOrchestrator에서 import해야 하며 [27], 배포 시 해당 파일이 누락되지 않도록 패키징에 주의해야 합니다. 또한 동적 임계값이 도입되면 **기존 테스트 케이스**들이 실패할 수 있습니다. 예컨대 이전에는 파일 21개 변경이 항상 실패였겠지만, 이제는 맥락에 따라 통과될 수도 있으므로, **품질 게이트 관련 단위테스트**를 업데이터하거나 새로운 기준에 맞게 작성해야 합니다 [28].

임계값 계산식의 **적절성**도 위험요소입니다. 공식이 너무 느슨하면 품질 저하를 초래하고, 너무 엄격하면 개선 적용이 불필요하게 막힐 수 있습니다. 따라서 **다양한 시나리오에 대한 테스트**와 조정이 필요합니다 [28]. 계획 문서에서는 이 작업의 위험을 Medium으로 평가하며, **임계값 계산 검증**에 특히 신경 써야 함을 언급하고 있습니다 [24] [29].

마지막으로, `verification_rounds` 등의 새로운 개념을 도입한 만큼, MetaOrchestrator의 **워크플로우 로직**도 약간 손봐야 합니다. 예를 들어 2회 검증이 요구된 경우, 한 싸이클 더 Improvement 과정을 반복하거나 추가 검사(다른 에이전트 호출 등)를 수행해야 할 것입니다. 이 부분은 LangGraph 기반 상태 전이 (섹션 3에서 후술)로 관리하면 깔끔하게 구현할 수 있습니다.

개선 4: 형식 온톨로지(Formal Ontology) + QualityAgent 도입

- **문제 진단:** 현재 시스템에는 관계 간 **형식 논리적 제약**(예: 추이성, 대칭성, 반사성, 비대칭 등)이 정의되어 있지 않아, **의미적으로 잘못된 관계**도 걸러지지 않고 넘어갈 위험이 있습니다 [30]. 또한 **QualityAgent**가 부재하여, 관계 출력에 대한 검증이 문자열 포맷 등의 표면적인 수준에 국한되고 있습니다 [30]. 그 결과 "Calculus → Addition"처럼 **논리적으로 부조리한 관계**도 통과할 수 있는 문제가 지적되었습니다 [31].

- **코드 변경 사항:** 이 문제를 해결하기 위해 **형식 온톨로지 모듈**과 **QualityAgent 모듈**을 새로 추가하고, RelationshipDefiner 및 Orchestrator 흐름에 이를 통합합니다.

(A) 관계 형식 온톨로지 정의 – `agents/relationship_ontology.py` 신규 파일을 만들어, 시스템이 다루는 관계 유형들을 열거하고 각 타입별 **논리적 성질(FormalProperties)**을 지정합니다 [32] [33]. 예를 들어 12가지 관계 유형(사전 정의된 taxonomy v0.2 기준)을 Enum으로 정의하고, 이에 대해 `FormalProperties` (dataclass)로 다음 다섯 가지 논리 속성을 기록합니다 [33]:

- `transitive` (추이적인지 여부)
- `symmetric` (대칭적인지 여부)
- `reflexive` (자기 자신에게 적용 가능한지 여부)
- `anti_symmetric` (만약 양방향 존재하면 동일 개체로 간주되는 관계인지 여부)
- `acyclic` (해당 관계로 **순환(loop)**이 허용되지 않는지 여부)

각 관계에 대해 위 속성을 설정함으로써, 예컨대 **"prerequisite"** 관계는 `transitive=True, symmetric=False, reflexive=False, anti_symmetric=True, acyclic=True`로 정의합니다 [34]. 이는 "선행 지식" 특성상 A→B, B→C이면 A→C (추이적)일 수 있고, 대칭은 아니며, 자기 자신을 선행요건으로 가질 수 없고, A→B와 B→A가 동시에 참이면 A와 B는 같은 개념이어야 하므로 안 되며, 순환이 발생해선 안 된다는 의미입니다. 반면 **"co-requisite"** (공동 전제)나 **"inverse-operation"**(역연산) 같은 관계는 `symmetric=True`로 명시하여, A→B면 B→A도 같은 관계로 성립되어야 함을 표시합니다 [35] [36]. 또 **"equivalence"**(동치)처럼 **완전대칭**이고 **추이적**이며 **모든 개체가 자기 자신과 동치**인 관계도 설정할 수 있습니다 [37]. (추가로 연구 보고서에서 제안된 새로운 관계 유형이 있다면, 예: "property-of", "peer-of" 등, 해당 온톨로지에 속성을 정의해주어야 합니다. 예컨대 "peer-of"는 대칭이어야 하고, "property-of"는 순환 금지 등을 적용해야 한다고 언급됩니다 [5].)

위와 같이 정의한 온톨로지를 활용하여, `validate_relationship_logic(rel_type, source_id, target_id, graph)` 함수를 구현합니다. 이 함수는 주어진 관계 타입과 소스/타겟 개념을 받아, 온톨로지 규칙에 비추어 **위반 사항**이 있는지 검사하고, 문제가 있을 경우 `ValidationError` 객체들을 리스트로 반환합니다 [38] [39]. 검사의 주요 내용은 다음과 같습니다:

- **Reflexivity 체크:** 만약 소스와 타겟이 같은 개념인데 해당 관계가 `reflexive=False`로 정의되어 있다면 오류를 리포트합니다 (자기 자신에게 적용 불가한 관계인데 **source==target**인 경우) [40].
- **Acyclic 체크:** 온톨로지에서 `acyclic=True`인 관계에 대해서는, 현재 **기존 그래프**를 참조하여 이번에 이 관계를 추가하면 **순환**이 생기는지 확인합니다. 구체적으로 DFS(깊이 우선 탐색)를 통해 **타겟에서 출발하여 소스로 도달 가능 여부**를 탐색함으로써 사이클 존재를 검사합니다 [41] [42]. 만약 새로운 관계 추가 시 cycle이 생긴다면 오류로 보고합니다 (예: 선행 지식 관계에서 A←…←B←A 형태 발견 시 오류) [43].
- **Symmetry 체크:** 온톨로지에서 `symmetric=True`로 정의된 관계에 대해서는, 현재 그래프에 **역방향 관계**(B→A)가 존재하는지 찾습니다 [44]. 없을 경우 이는 **대칭 관계가 완비되지 않은 상태**이므로 경고를 발생시킵니다. (예: A와 B는 동등한 개념이어야 하는 관계인데 A→B만 있고 B→A가 없다면 경고) [44]. 반대로, `anti_symmetric=True` 관계의 경우 **역관계가 존재하면 논리적 모순**이므로 이것도 오류로 처리해야 합니다. (계획 코드에는 반대 방향 존재 시 어떻게 처리하는지 명시적 예시는 없지만, anti_symmetric 특성이 True인 관계에서 `find_relation(B, A)`가 발견되면 이는 곧잘 **정합성 위배**로 볼 수 있습니다. 구현 시

이러한 케이스도 `ValidationError("ERROR", "antisymmetry_violation", ... )` 형태로 추가해야 합니다.)

위 검증 함수가 반환한 오류/경고 리스트는 ValidationError 객체의 문자열 표현으로 만들어집니다 [44]. 예를 들어 순환 오류의 메시지는 "[ERROR] circular_dependency: Adding A→B creates a cycle ..." 형태가 될 것이고, 대칭 경고는 "[WARNING] symmetry_incomplete: ... reverse B→A missing" 식으로 나타납니다.

(B) QualityAgent 구현 – `agents/quality_agent.py` 파일을 신설하여, **지식 그래프의 품질을 종합적으로 점검**하는 에이전트를 구현합니다. QualityAgent는 주로 세 가지 검사를 수행할 수 있어야 합니다 [45]:

1. **구문 검증 (Syntax)** – 출력된 지식 그래프 (개념들과 관계들의 JSON 구조)에 필수 필드 누락이나 포맷 오류가 없는지 확인합니다. 예를 들어 `concepts` 리스트나 `relationships` 리스트 존재 여부, 각 관계에 `source_concept_id`, `target_concept_id`, `relationship` 키가 있는지 등을 체크하고 누락 시 오류를 기록합니다 [46] [47].
2. **의미 검증 (Semantics)** – 도메인 지식에 비추어 부자연스러운 관계가 없는지 검사합니다. 예컨대 소스/타겟 개념이 그래프에 실제로 존재하는지 확인하고 [48], 추가로 교육 도메인에서 의미상 이상한 관계를 탐지합니다. 계획 코드 예시로는 선행 지식 관계의 경우 학년(grade level)을 비교하여, **A가 B의 선행인데 A의 학년 수준이 더 높다면 경고**하는 로직이 포함되어 있습니다 [49]. 이처럼 분야별 상식 검증을 통해 잠재적 오류 관계를 솎아냅니다.
3. **온톨로지 준수 검증 (Ontology compliance)** – 앞서 구현한 `validate_relationship_logic` 함수를 모든 관계에 대해 실행하여 형식 논리 위반 여부를 확인합니다 [50] [51]. 각 관계에 대해 RelationType을 파싱하고 (미등록 타입이면 오류) [52], 온톨로지 검증을 돌린 뒤 반환된 에러/워닝들을 종합 리스트에 추가합니다 [53] [54].

QualityAgent의 `validate_knowledge_graph(graph)` 메서드는 이처럼 **다층적인 검증**을 거쳐, 최종적으로 `{"errors": [...], "warnings": [...], "passed": True/False}` 형식의 결과를 돌려주도록 설계합니다 [55] [56].

- **연계 및 충돌 요소:** RelationshipDefiner와 MetaOrchestrator에 **QualityAgent/ontology 연계 로직**을 삽입해야 합니다. 우선 RelationshipDefiner에서 **새 관계를 판별할 때 즉시 온톨로지 검증**을 수행하도록 개선할 수 있습니다. 개선 계획에서는 `relationship_definer.py`에 `validate_relationship(self, relationship: Dict) -> Dict` 메서드를 추가하여 개별 관계 출력에 대해 온톨로지 체크를 수행하는 예시를 제시했습니다 [57] [58]. 이 메서드는 주어진 관계 딕셔너리에서 타입을 추출해 RelationType Enum으로 파싱하고, 바로 `validate_relationship_logic`을 호출하여 오류가 있으면 `{"validation": "FAILED", "errors": [...]}`를 리턴하도록 합니다 [59] [60]. RelationshipDefiner가 하나의 관계를 판별할 때마다 이 함수를 불러 **즉각적인 피드백**을 얻어낼 수 있습니다. 다만 이때 `validate_relationship_logic`에 **현재까지 구축된 그래프 구조** (`self.existing_graph`)를 인자로 넘겨주어야 순환검사가 정확히 이뤄집니다 [60] [43]. 그러므로 RelationshipDefiner 내부에 현재까지 추가된 관계들을 누적 추적하는 `existing_graph` 상태를 유지하거나, Orchestrator 차원에서 관리하는 **글로벌 그래프 구조**를 참조할 필요가 있습니다 (Graph 상태 관리에 대한 내용은 위험요소 및 LangGraph 활용 부분에서 다시 언급).

한편, 개선 작업의 큰 흐름에서 보면 **QualityAgent를 언제 작동**시킬지가 중요합니다. RelationshipDefiner 단계에서 1차적으로 검증을 하더라도, 최종 통합 전에 **전체 그래프에 대한 종합 검사**를 위해 Orchestrator가 마지막에 QualityAgent를 실행하는 것이 바람직합니다. 예를 들어 Phase 2 종료 시 (모든 관계를 생성하고 난 후) `quality_agent.validate_knowledge_graph(full_graph)`를 호출하고, 만약 `passed=False` (오류 존재)인 경우 해당 오류들을 정리하여 **개선 적용을 보류**하거나 SocraticMediator를 통해 문제를 되짚도록 할 수 있습니다.

이 통합으로 인한 충돌 요소로는, **그래프 상태 관리의 복잡성 증가**가 있습니다. 순환 검출 등의 검증을 위해서는 **관계 추가 순서에 따라 실시간으로 그래프를 업데이트**해야 합니다. 즉, RelationshipDefiner가 하나의 관계를 판정할 때마다

임시 그래프에 추가하고, QualityAgent/ontology 검증을 거쳐 실패하면 롤백하거나 수정하는 흐름이 될 수 있습니다. 이는 구현 난이도가 높고 알고리즘적으로도 비용이 들지만, **논리적 완결성**을 확보하기 위해 필수적인 과정입니다 [61] [62]. 이러한 복잡성을 관리하기 위해 제안되는 것이 **LangGraph 기반 상태 제어**이며, 다음 섹션에서 다룰 예정입니다.

또 다른 고려사항은 **관계 타입의 확장 및 유지보수**입니다. 형식 온톨로지 모듈에 관계 타입 Enum과 속성을 하드코딩했기 때문에, 추후 새로운 관계 종류를 도입하거나 정의를 변경하려면 이 코드를 수정해야 합니다. 이를 좀 더 유연하게 개선하려면 **온톨로지 스키마**를 외부 JSON/YAML로 정의하고 로딩하는 방식을 도입하거나, DB나 KG에서 온톨로지를 가져오도록 할 수도 있습니다. 하지만 초기 단계에서는 하드코딩이 명료하므로 우선은 이 형태로 구현하고, 향후 **taxonomy 확장 시 관리 비용**을 인지해야 합니다.

Risk & Mitigation: 계획 문서는 이 부분을 **High Risk**로 분류하였습니다 [63]. 새로운 모듈들이 다수 생기고, 그래프 순환검사처럼 **완전히 새로운 로직**이 들어가기 때문입니다. 따라서 **철저한 단위 테스트**와 시뮬레이션이 필요합니다. 특히, 검증 로직이 실제 문제를 잘 잡아내는지 (False negative가 없는지)와 반대로 유효한 관계를 오검출(false positive)하지 않는지 살펴야 합니다. 예를 들어 대칭 관계 경고는 허용하되, 치명적 오류는 막아야 하는데 이 균형을 잘 맞추어야 합니다. 또한 경고/오류의 **메시지가 개발자나 도메인 전문가에게 충분히 이해 가능**하게 구성되어야, HITL 단계(다음 개선안)에서 사람이 판단을 내리기 쉽습니다. QualityAgent는 **내부 도구이므로 성능보다는 철저함이 우선**이지만, 그래프 규모가 커질 경우 DFS 순환검사 등의 비용도 고려해야 합니다. 필요시 효율을 위해 그래프 인접리스트 캐시를 재활용하거나, **네트워크x** 같은 라이브러리를 활용하는 것도 검토 가능합니다.

- **예상 효과:** 이 개선을 통해 시스템은 **논리적으로 일관된 지식 그래프**만을 허용하게 됩니다. 예컨대 이제 **선행 관계의 순환**은 자동 탐지되어 차단되고, **대칭 관계의 짝이 누락되는 문제**도 경고를 통해 인지할 수 있습니다. 수학 교육 도메인에 특화된 의미 검증(학년 불일치 경고 등)도 포함됨으로써, 생성된 관계망의 **교육적 타당성**을 제고할 수 있습니다. 이러한 품질 향상은 추후 루프를 통해 새로운 관계를 추가하거나 수정할 때도 **안전망**으로 작용하여, 시간이 지날수록 지식 그래프의 **정합성**을 유지할 것으로 기대됩니다 [64].

개선 5: HITL (Human-in-the-Loop) 프레임워크 도입

- **문제 진단:** 현재 자가개선 루프의 모든 개선 조치가 **자동으로 적용**되고 있으며, 중간에 사람이 개입하여 검토하거나 승인을 요청하는 단계가 없습니다. 이는 **고위험 변경사항**의 경우 문제가 될 수 있습니다 [65]. 예를 들어 새로운 관계 타입을 추가하거나, 매우 중요한 개념의 관계를 대거 수정하는 결정은 도메인 전문가의 판단을 거치는 것이 안전하나, 현재는 그러한 **검수 절차** 없이 Claude 모델의 판단대로 바로 적용됩니다. 이로 인해 잘못된 변경이 시스템에 전파되거나, 중요한 개념 맥락이 어그러질 위험이 있습니다 [65].

- **개선 방향: Human-in-the-Loop** 체크포인트를 시스템 워크플로우에 도입하여, **중요 의사결정 지점에서 인간의 승인 또는 피드백을 요구**하도록 합니다. 이를 위해 새로운 모듈 `agents/hitl_checkpoints.py`를 생성하고, MetaOrchestrator에 후크를 거는 형태로 통합합니다 [66]. 구현 측면에서, MetaOrchestrator가 개선 루프를 진행하다가 **특정 조건**을 만나면 (예: 품질 게이트 통과했지만 경고가 있는 경우, 새로운 관계 타입이 추가된 경우, 선행 지식 그래프에 큰 변화가 있는 경우 등) HITL 모듈에 정의된 함수를 호출하도록 합니다. 이 함수는 변경 사항 요약, 관련 메트릭, QualityAgent 경고/오류 등을 정리하여 **인간에게 표시**하고, **승인 대기 상태**로 진입시킵니다. 인간 검토자는 예컨대 웹 대시보드나 CLI 프롬프트를 통해 변경을 확인하고 "승인", "재시도 요청", "취소" 등의 결정을 내릴 수 있을 것입니다. 결정이 내려지면 HITL 모듈은 Orchestrator에 신호를 보내 루프를 재개합니다 (승인 시 계속 진행, 재시도 시 개선안을 폐기하고 다음 아이디어로 진행 등).

- **구체적 설계 예시:** Pseudo-code로 표현해 보면, MetaOrchestrator 워크플로우 내에 다음과 같은 구조가 삽입될 수 있습니다:

```
result = improvement_agent.propose_change(...)
if result.risk_level == "HIGH" or result.change_type == "new_relationship_type":
    from agents import hitl_checkpoints
    user_decision = hitl_checkpoints.request_approval(result.summary)
```

```
if user_decision == "APPROVE":
    apply_change(result)
elif user_decision == "REJECT":
    skip_change(result)
elif user_decision == "REVISE":
    result = improvement_agent.revise_change(...)
    ... (다시 HITL 또는 자동 진행)
```

위 예시에서 `propose_change`는 Self-Improver 에이전트 등이 생성한 개선안이며, 사전에 위험도를 분류했다고 가정합니다. HITL 조건은 시스템 요구에 따라 다양하게 정의할 수 있습니다. 예를 들어 **Phase 3**의 핵심 목표가 "중대한 결정에 100% 인간 감독"이므로 [67] [68], 위험도가 High로 분류된 개선 (#4 온톨로지/QualityAgent 관련, #5 HITL 자체 도입, #3 동적 게이트도 어느 정도)이나 **"검증 라운드 2회 요구"** 경고가 나온 경우 등을 후보로 삼을 수 있습니다.

- **통합 및 고려사항:** HITL 프레임워크 도입은 본질적으로 **시스템의 동작 방식을 동기에서 비동기**로 바꾸는 부분이 있어 유의해야 합니다. 사람이 응답을 할 때까지 시스템이 대기해야 하므로, UI와의 연결이나 시간 초과 전략 등이 필요합니다. 만약 실시간 인터랙션이 어렵다면, **오프라인 승인 절차**(예: PR 리뷰처럼 개선안을 기록해두고 사람이 나중에 검토)로 운영하는 방법도 있습니다. 이 경우 루프는 일단 자동으로 진행하되 최종 반영 전에 대기 상태로 멈추고, 별도 관리자 인터페이스에서 승인 처리가 이루어지면 최종 적용하는 식입니다.

충돌 지점으로는, 이러한 HITL 대기가 **전체 시스템 처리량을 저해**할 수 있다는 점입니다. 개선안이 자주 발생하면 사람의 연속된 승인이 필요해 병목이 될 수 있으므로, **임계값**을 잘 정해야 합니다. 예컨데 사소한 변경까지 전부 승인 받도록 하면 비효율적이니, **정말 중요한 변화에 한해** 적용하도록 임계 조건을 선별해야 합니다.

또한 HITL 단계에서는 사람이 개입하므로 **UI/UX 설계**와 **로그/이력 관리**도 중요합니다. 어떤 변경을 누가, 언제 승인했는지 기록하고, 추후 추적 가능하게 하는 기능이 필요합니다. 시스템 관점에서는, HITL 모듈이 돌려준 결정에 따라 Orchestrator의 상태를 변경해야 하므로, 상태 전이 관리가 복잡해집니다. (LangGraph 도입 시 이러한 대기/재개 상태를 노드로 표현하여 관리하면 한결 명확해질 것입니다.)

- **종합 의견:** HITL 도입은 위험도가 높은 작업이지만 [69], **시스템 신뢰성**을 확보하는 데 큰 도움이 될 것으로 보입니다. 특히 교육 도메인에서는 현장의 전문가 검토가 중요하기 때문에, 자동 생성된 관계망을 교사가 확인하고 승인하는 과정을 거친다면 결과물에 대한 **신뢰도**와 **도메인 적합성**이 크게 향상될 것입니다. 이 기능은 다른 개선들(온톨로지, 품질게이트 등)과 시너지를 이루어, **자동화 + 인간감수**라는 이중 안전장치를 구축하게 됩니다. 구현 시에는 우선 작은 범위(예: 새로운 관계 유형 추가 시에만 HITL 요청)부터 적용하고, 안정성을 확인한 후 점진적으로 확대하는 **단계적 도입 전략**이 권장됩니다.

개선 6: SocraticMediatorAgent – 소크라틱 Q&A 병렬화

- **문제 진단:** Socratic Mediator 에이전트 (`agents/socratic_mediator_agent.py`)는 Claude에게 복잡한 문제를 해결할 때, 여러 하위 질문을 단계별로 던지고 그 답을 종합하는 **대화형 진단**을 수행합니다. 그러나 현재 구현은 **질문들을 순차적으로 처리**하는데, 이로 인해 전체 응답 지연(latency)이 누적되는 문제가 있습니다. 예컨대 3개의 하위 질의가 있으면 3번 연속 LLM 호출을 해야 하고, 각 호출이 5초라면 총 15초가 소요되는 식입니다. 개선 계획에 따르면 이러한 순차 호출로 **최대 90%의 시간 페널티**가 발생하고 있다고 분석되었습니다.

- **개선 방향: 병렬 실행**을 도입하여 Socratic Q&A 단계의 지연을 단축합니다. 이를 두 가지 측면에서 고려할 수 있습니다:

- **코드 레벨 병렬화:** Python의 concurrency 기법(예: `asyncio`, `threading`, `concurrent.futures`)을 활용하여 **여러 하위 에이전트 호출을 동시에** 수행합니다. 예를 들어

Mediator가 동시에 여러 정보를 수집할 수 있는 경우 (`knowledge_agent`에 A와 B를 동시에 질의, 또는 여러 문서를 병렬 검색), 이를 병렬 태스크로 실행하고 `gather()`로 결과를 모읍니다. CPU 연산이 아니라 I/O(LM API 호출) bound이므로 쓰레드풀이나 async를 사용하면 거의 선형에 가깝게 시간 단축이 가능합니다. 단, 공유 자원 접근 시 **경쟁 상태(race condition)**가 없도록 주의하고, 결과 병합 순서를 보장해야 합니다. 이 방식은 코드 구현의 복잡도를 높이므로 철저한 테스트가 필요하며, 계획서에서도 **Risk: Medium (병렬 패턴 테스트 필요)**로 평가되었습니다.

- **프롬프트 레벨 병렬화:** Claude에게 한 번의 호출로 여러 질문을 던져 **병렬 사고**를 유도하는 방법입니다. 이는 예를 들어 시스템 프롬프트에 "당신은 두 명의 독립적인 분석자처럼 행동하라. 질문1과 질문2를 동시에 고려하여 각각 답하고 결론을 종합하라." 같은 지시를 추가하는 것입니다. 개선 계획에서는 구체적으로 "prompt (line 60-69) with parallel Task examples"를 넣으라고 언급되어 있는데, 이는 LLM이 한 턴에 다중 질의를 병렬적으로 다루는 시나리오를 학습시키는 프롬프트 기법으로 보입니다. 예컨대 이전 대화 예시로 두 가지 질문을 한꺼번에 하고 각각 답변하는 포맷을 보여줌으로써, 모델이 순차 응답 대신 병렬적 사고흐름을 모방하도록 할 수 있습니다. 이 접근은 **모델의 응답 구조**를 바꾸는 것이어서 설계가 까다롭지만, 성공하면 단일 호출로 여러 답을 얻어내므로 지연을 크게 줄일 수 있습니다.

- **코드 구현 예시:** 코드 레벨 병렬화를 구현한다면, 파이썬 코드로는 다음과 같은 형태가 될 것입니다:

```
import asyncio
async def ask_multiple(questions):
    # questions: [(agent, prompt), ...] 리스트
    tasks = []
    for agent, prompt in questions:
        tasks.append(asyncio.create_task(agent.ask(prompt)))
    results = await asyncio.gather(*tasks)
    return results

# 사용 예시
sub_questions = [
    (knowledge_agent, "개념 A의 정의와 속성은?"),
    (knowledge_agent, "개념 B의 정의와 속성은?")
]
answers = asyncio.run(ask_multiple(sub_questions))
# answers 리스트에는 A와 B에 대한 정의 결과가 병렬 수집됨
```

위 예시는 동일한 `knowledge_agent`를 병렬 호출하는 형태지만, 실제 SocraticMediator에서는 서로 다른 도구(예: 위키 검색 에이전트와 코드 실행 에이전트 등)를 병렬 쓰는 경우도 있습니다. 중요한 점은 **결과를 한데 모아 종합하는 로직**입니다. 병렬화 이전에는 직렬 흐름이므로 각 단계 결과를 다음 입력에 활용하기 쉬웠으나, 병렬화한 경우 한 번에 모인 결과를 **어떻게 결합하여 다음 질문이나 최종 답변에 반영**할 지 설계해야 합니다. 예컨대 A, B 정의를 동시에 받아서 둘을 비교하는 작업은, 두 결과 모두 확보된 이후에야 가능하므로, 병렬 수집 후 **동기화 지점(synchronization point)**을 명확히 두어야 합니다.

- **잠재적 충돌과 고려사항:** 쓰레드나 async를 도입하면 **디버깅 난이도**가 올라가고, 예외 처리도 복잡해질 수 있습니다. 각 하위 작업에서 예외(시간 초과 등)가 발생하면 전체 gather가 취소되지 않도록 적절히 shield하거나, 타임아웃을 개별 적용해 일부 실패 시 나머지 결과로 계속 진행하는 전략도 필요합니다. 또한 OpenAI API 등 LLM 백엔드의 **동시 요청 제한** 또는 **요금** 문제도 고려해야 합니다. 한 번에 5개의 요청을 병렬로 보내면 토큰 사용량 피크가 높아지고 비용도 증가할 수 있으므로, **병렬화 정도를 적절히 제한**해야 합니다.

프롬프트 차원의 병렬화는 모델이 **형식적 논리**를 혼용할 위험이 있습니다. 여러 질문을 동시에 다루는 사례가 학습에 부족하면, 모델이 답변을 혼동하거나 구조화에 실패할 수 있습니다. 이 부분은 여러 시도를 통해 프롬프트 최적화를 해야 하며, 결국 일정 부분 성능 확보를 위해 코드 레벨의 실제 병렬 실행이 함께 필요할 수 있습니다.

- **효과:** 올바로 구현된다면, SocraticMediator 단계에서 **최대 90%까지 응답 시간 단축**을 기대할 수 있습니다. 이는 사용자 경험 측면에서 매우 큰 개선입니다. 특히 여러 자료원에서 정보를 모아 결합하는 작업이 빨라지므로, 최종적으로 **자가개선 루프 전체의 처리 시간이 단축(-90%)**될 수 있습니다 [70] [71]. 다만, 이러한 최적화는 모델의 **사고 품질을 떨어뜨리지 않는 선**에서 이루어져야 합니다. 즉, 병렬화로 인해 모델이 충분한 고찰 없이 성급히 결론을 내리는 부작용이 없도록, **Mediator의 후속 로직에서 검증 및 보완 질문을 할 수 있도록 유지**해야 합니다.

以上가 6가지 핵심 개선 항목에 대한 코드 수준 검토입니다. 다음으로, 이러한 개선 중 **프롬프트/출력 형식 변경**, **LangGraph 도입**, **QualityAgent 검증 로직 세부 구현**, **ExternalValidator 설계**, **성능/비용/의존성** 측면을 개별로 추가 분석합니다.

2. 프롬프트 및 JSON 출력 스키마 변경 (예시 코드 포함)

앞서 개선 항목들 중 **RelationshipDefiner (개선 1, 2)**와 **SocraticMediator (개선 6)**는 **프롬프트 내용 수정**과 **출력 JSON 구조 변화**를 수반합니다. 이러한 변화가 실제 코드에 어떻게 반영되어야 하는지를 예시와 함께 정리합니다.

- **시스템 프롬프트 수정:** RelationshipDefiner의 프롬프트에 불확실성 분석 섹션과 Chain-of-Thought 단계 지침을 추가하는 것은 앞서 설명한 대로입니다 [2] [13]. 이를 구현하려면 코드 상에서 프롬프트 문자열을 구성하는 부분에 해당 텍스트 블록을 삽입해야 합니다. 예컨대 `relationship_definer.py` 내 프롬프트 정의부에서 기존 "OUTPUT FORMAT" 섹션 이전에 CoT 안내문을 넣고, "UNCERTAINTY ANALYSIS" 섹션을 적절한 위치에 삽입합니다. 이러한 프롬프트 문자열은 종종 파이썬 멀티라인 문자열(`"""`)로 작성되어 있으므로, **정확한 들여쓰기와 형식**을 유지해야 합니다. CODE-IMPROVEMENT-PLAN.md에 프롬프트 변경 예시가 상세히 나와 있으므로 이를 참고하여 반영하면 됩니다 [2] [13].

- **JSON 출력 스키마 확장:** RelationshipDefiner의 응답 JSON에 새로운 필드들이 추가되므로, **출력 데이터 클래스를 사용하는 경우 업데이트**해야 합니다. 예를 들어 `Relationship`라는 dataclass가 있었다면 `uncertainty_breakdown: Dict[str, float]`, `uncertainty_reason: str`, `alternative_classifications: List[Dict]` 등을 필드로 추가해야 할 것입니다. 만약 단순히 딕셔너리를 사용하는 구조라면 특별한 변경 없이도 키-값을 추가 저장하면 되지만, 그 값을 어디까지 활용할지 정해야 합니다 (예: SelfImproverAgent가 alternative_classifications를 이용해 다른 액션을 취할지 여부).

JSON 스키마 변경의 대표적인 예시는 개선 1에서 다룬 **불확실성 필드 추가**입니다. 위에 제시한 JSON 코드블록 [1]처럼, `confidence` 이외에 nested object와 리스트가 들어가는 구조이므로, **(de)serialization 로직**이 있는 경우 해당 키를 인식하도록 갱신해야 합니다. 특히 `uncertainty_breakdown`의 sub-keys(epistemic 등)를 객체로 매핑하거나, alternative_classifications를 리스트→객체로 변환하는 처리 등이 필요할 수 있습니다.

- **응답 파싱 로직 업데이트:** 개선 2의 CoT 도입으로 인해 **LLM 응답 파싱 로직**이 복잡해진 부분은 이미 예시 코드를 제시했습니다 [15] [16]. 이를 실제 코드에 반영할 때 주의할 점은, 모델 응답이 항상 예시처럼 나오지 않을 수 있으므로 **방어적 코딩**을 해야 한다는 것입니다. 예를 들어 Phase 구분자가 누락되거나 JSON 블록이 `표시 없이 끝날 경우 등 예외 상황 처리분기도 추가해야 합니다. 개선 계획 예시에서는 단순히 ``elif "`" in response_text: ... pass``로 표시했지만, 실제 구현에서는 이 부분에 **이전 버전 파싱 방식**(CoT 없던 시절)을 넣어주어야 할 것입니다 [72].

또 하나 고려할 점은, **reasoning_trace를 어느 범위로 붙일지**입니다. 현재 예시는 모든 관계에 동일한 reasoning_trace를 달지만, 만약 모델이 여러 관계 각각에 대해 개별 reasoning을 달리 기술하는 형태로 확장한다면

파싱을 더 세분화해야 합니다. (예: Phase 1을 관계별 소단락으로 구성). 현재 단계에서는 **전체 관계 세트에 대한 단일 reasoning_trace**로 충분해 보입니다.

- **병렬 Q&A 프롬프트 예시:** 개선 6에서 언급한 "parallel Task examples"을 프롬프트에 포함하는 작업은 조금 특수한 경우입니다. 명시적인 코드는 없지만, 가령 프롬프트에 다음과 같은 예시를 넣을 수 있을 것입니다:

```
### 예시: 병렬 Socratic 질문
질문: "X 개념과 Y 개념의 관계를 분석하시오."
생각:
- Task1: X의 정의와 배경 조사 (동시에 수행)
- Task2: Y의 정의와 배경 조사 (동시에 수행)
- 두 결과 종합하여 관계 추론
답변: {"relationship": ...}
```

위와 같은 형식으로 모델이 **생각 단계에서 병렬로 두 작업을 나열**하는 예시를 보여주면, 실제 질문에서도 Task1, Task2를 병렬 진행하려 할 수 있다는 발상입니다. 이를 구현하려면 `socratic_mediator_agent.py`의 프롬프트 빌드 부분에 해당 예시를 추가합니다. 다만 이 기법의 효과는 미지수이므로, 모델 출력 양상이 원하는 대로 나오는지 확인이 필요합니다.

- **예시 코드 적용:** 요약하면, 프롬프트와 출력 스키마 변경에 따른 대표 코드 조각들은 이미 개선 계획 문서에 제시된 것을 참고하여 그대로 혹은 약간 수정하여 적용하면 됩니다. 앞서 인용한 JSON 확장 예 [1] 와 파싱 코드 예 [15] [16] 가 그 예입니다. 이러한 변경을 반영한 뒤에는 **유닛 테스트**를 작성하여, LLM 응답 예시를 입력했을 때 파싱 결과가 올바른지, 새로운 필드가 잘 들어오는지 등을 검증해야 합니다. 특히 JSON 스키마의 변경은 Downstream 모듈에 영향이 크므로, **통합 테스트**를 통해 SocraticMediator → RelationshipDefiner → SelfImprover 전체 흐름에서 데이터가 문제 없이 전달되는지 점검해야 합니다.

3. LangGraph 도입에 따른 구조 리팩토링 및 상태 전이 설계

LangGraph는 멀티 에이전트 LLM 시스템에서 **상태 머신 기반의 워크플로우 오케스트레이션**을 가능하게 하는 프레임워크입니다 [73]. 현행 Claude 자가개선 루프는 MetaOrchestrator가 **절차적 코드**로 일련의 에이전트 호출과 분기 로직을 수행하고 있는데, LangGraph를 도입하면 이를 **명시적인 상태 전이 그래프** 형태로 재구성할 수 있습니다. 즉, 각 단계(노드)는 하나의 에이전트 동작 또는 체크포인트를 의미하고, 조건에 따라 다음 단계(에지)로 전이하는 방식입니다 [74] [75]. 이러한 재구성은 코드의 **가독성과 유지보수성**을 높이고, 예외 처리나 재시도 로직을 구조적으로 관리할 수 있게 해줍니다.

- **현재 구조의 한계:** MetaOrchestrator의 절차적 흐름에서는 여러 if/else 분기가 중첩되고, loop나 재귀 호출 등으로 복잡성이 증가하는 경향이 있습니다. 특히 이번 개선안들을 적용하면, 품질 게이트 실패 시 재시도, HITL 승인 대기, verification_rounds 루프 등 **상태**가 늘어나서, 절차 코드를 이해하기 어려워질 수 있습니다. 또한 새로운 에이전트가 추가될 때마다 Orchestrator 코드를 수정해야 하므로 **결합도**가 높습니다.

- **LangGraph 적용 리팩토링:** 먼저 각 주요 단계를 **State 노드**로 정의합니다. 예를 들어 다음과 같은 상태 목록을 구상할 수 있습니다:

상태 (State 노드)	설명 (해당 에이전트/작업)
Start	시작 상태 (초기 입력 로드 등)
DefineRelationship	RelationshipDefiner 에이전트 실행 – 개념 관계 추론
RefineQuestion	SocraticMediator 실행 – 불확실성 높은 경우 추가 질의 및 프롬프트 보강

상태 (State 노드)	설명 (해당 에이전트/작업)
QualityCheck	QualityAgent 실행 – 논리 검증 및 품질 게이트 평가
HumanReview	HITL 체크포인트 – 인간 승인 대기
ApplyChange	개선사항 적용 – 최종적으로 지식 그래프/코드에 변경 반영
End	종료 상태 (루프 완료)

각 상태에 대해 **상태 처리 함수**를 바인딩합니다 (LangGraph에서는 `add_node("상태명", 함수)` 형태로 구현합니다 [76]). 예를 들어 `"DefineRelationship"` 상태에는 실제 `relationship_definer.run(input)`을 호출하는 함수를 연결합니다. `"QualityCheck"` 상태 함수는 quality_agent.validate_knowledge_graph를 실행하고 그 결과를 토대로 품질 게이트 통과여부와 경고를 반환합니다.

그런 다음 **상태 전이 규칙(transition edges)**을 정의합니다 [77]. 전이는 어떤 상태가 끝났을 때 **다음으로 이동할 상태를 결정하는 조건**입니다. LangGraph에서는 전이 정의시 조건부 함수를 사용하거나, 상태 처리 함수 자체가 다음 상태명을 반환하게 할 수도 있습니다. 본 시스템에 맞게 전이를 설계하면 예를 들면 다음과 같습니다:

- Start → DefineRelationship : (무조건) 입력이 주어지면 바로 관계 정의 단계로 이동.
- DefineRelationship → RefineQuestion : 만약 RelationshipDefiner 결과에 `confidence < 0.85`이거나 `uncertainty_reason`이 존재하면 이 분기로 이동. 즉 **불확실성이 높을 때** SocraticMediator로 **추가 질의/보강** 상태로 간다 [78].
- DefineRelationship → QualityCheck : 반대로 충분히 확실한 관계가 도출되었으면 바로 품질 검증 단계로 진행.
- RefineQuestion → DefineRelationship : SocraticMediator가 추가 정보를 얻어 SelfImprover가 프롬프트를 개선했다면, 다시 관계 정의를 **재시도**하도록 루프백.
- QualityCheck → HumanReview : QualityAgent/품질게이트 결과 **오류는 없으나 경고가 있거나**, 또는 **오류가 있지만 재시도 가능한 경우** (예: dynamic quality gate failed with retry_allowed=True [79] [80])일 때 인간 검토로 보냄. 특히 경고 중 “Mission-critical affected” 같이 매우 중요한 변화라면 사람 확인이 필수입니다.
- QualityCheck → DefineRelationship : QualityAgent 결과 오류가 있고 자동 재시도가 가능하다면 (retry_allowed), 원인에 따라 관계 재설정 또는 추가 정보 확보를 위해 앞 단계로 돌아갑니다. 예컨대 **논리적 오류**(순환 등)라면 해당 관계를 버리거나 수정해야 하므로, SelfImproverAgent를 통해 새로운 제안을 얻은 뒤 다시 관계 정의를 할 수 있습니다.
- QualityCheck → End (성공 경로): 검증 통과(passed=True)且경고 없음 등 **완전 통과**일 때 최종 적용 단계로 넘어가거나, 여기서는 간단히 End로 종료 처리.
- HumanReview → ApplyChange 또는 DefineRelationship : 사람이 **승인**하면 ApplyChange로 가서 결과를 반영하고 종료. 사람이 **거부/수정 요청**하면 개선안을 버리거나 조정하여 다시 관계 정의 단계로 돌아가거나, 또는 루프를 완전히 종료할 수도 있습니다.
- ApplyChange → End : 변경사항을 실제 적용하고 루프 종료.

위 전이들을 그림으로 표현하면 아래와 같습니다:

```
graph LR
  Start --> DefineRelationship
  DefineRelationship -->|불확실성 높음| RefineQuestion
  DefineRelationship -->|확실함| QualityCheck
  RefineQuestion --> DefineRelationship
  QualityCheck -->|재시도 필요| DefineRelationship
  QualityCheck -->|HITL 필요| HumanReview
```

```
    QualityCheck -->|통과| ApplyChange
    HumanReview -->|승인| ApplyChange
    HumanReview -->|거부/수정| DefineRelationship
    ApplyChange --> End
```

(도식: 상태와 전이 예시)

실제 LangGraph 코드로는 다음과 같이 정의할 수 있습니다 (개념적 pseudo-code):

```
graph = StateGraph()
graph.add_node("start", start_func)
graph.add_node("define_rel", define_relationship_func)
graph.add_node("refine_q", refine_question_func)
graph.add_node("quality_check", quality_check_func)
graph.add_node("hitl_review", hitl_review_func)
graph.add_node("apply_change", apply_change_func)
# 상태 전이 규칙 추가
graph.add_edge("start", "define_rel")
graph.add_edge("define_rel", "refine_q", condition=lambda state: state["uncertainty"] > 0.15)
graph.add_edge("define_rel", "quality_check", condition=lambda state: state["uncertainty"] <=
0.15)
graph.add_edge("refine_q", "define_rel")
graph.add_edge("quality_check", "hitl_review", condition=lambda state: state["need_human"]
== True)
graph.add_edge("quality_check", "define_rel", condition=lambda state: state["passed"] == False
and state["retry"] == True)
graph.add_edge("quality_check", "apply_change", condition=lambda state: state["passed"] ==
True)
graph.add_edge("hitl_review", "apply_change", condition=lambda state: state["approved"] ==
True)
graph.add_edge("hitl_review", "define_rel", condition=lambda state: state["approved"] == False)
```

※ 여기서 `state` 딕셔너리는 각 상태 처리 함수가 리턴하거나 갱신하는 공유 상태이며, `state["uncertainty"]` 등은 관계 정의 결과의 불확실성 정도, `state["need_human"]`은 품질체크 결과 인간 확인 필요 여부 등을 나타낸다고 가정했습니다.

- **리팩토링 효과:** 이와 같이 LangGraph로 구조를 변경하면, **새로운 단계 추가나 조건 변경이 용이**해집니다. 예컨대 추후 새로운 에이전트 (ExternalValidator 등)를 끼워넣고자 할 때 해당 상태와 전이만 정의하면 Orchestrator 코드의 다른 부분을 크게 건드리지 않고 통합할 수 있습니다. 또한 **상태 시각화 도구**나 **로그 추적**에 LangGraph를 활용하면, 에이전트들이 어떤 경로를 거쳐 실행되었는지 한눈에 파악할 수 있어 디버깅에 도움이 됩니다 (LangGraph는 종종 상태 다이어그램을 그리거나 실행 경로를 로깅하는 기능이 제공됩니다 [81]).

다만, LangGraph 도입에는 초기 리팩토링 비용이 있으며, 팀원들이 이 개념에 익숙해져야 하는 학습비용도 있습니다. 또한 성능 면에서는 미미하지만 추가 추상화 계층이 들어가므로 overhead가 약간 생길 수 있으나, 이는 현대 Python으로 무시해도 될 수준입니다. **결론적으로**, 시스템이 점점 커지고 복잡해지는 것을 대비한다면 LangGraph로의 전환은 유지보수성을 높이고 **오류를 예방**하는 투자라고 할 수 있습니다.

4. QualityAgent의 DFS 순환 검증 및 대칭성 검증 구현 방안

QualityAgent의 핵심 역할 중 하나는 **그래프의 순환 관계를 탐지**하고 **대칭 관계의 일관성**을 확인하는 것입니다. 이를 구현하기 위해 DFS(Depth-First Search) 알고리즘을 활용한 방법과, symmetry 검사 로직을 상세히 제안합니다 (개선 4에서 개괄적으로 설명한 내용을 보강).

- **순환 검증 (사이클 탐지):** 온톨로지에서 `acyclic=True`로 명시된 관계 유형 (주로 **선행 관계** 같은 위계적 관계)에 대해서는, **새로운 관계를 추가했을 때 사이클이 형성되는지 검사**해야 합니다. 가장 단순한 방법은 **DFS 또는 BFS**로 현재 그래프를 탐색하는 것입니다. 구체적으로, "A→B" 관계를 추가하려 할 때, 그래프 상에서 B를 시작 노드로 DFS를 수행하여 A에 도달 가능한지 확인합니다 [82] [42]. 도달하면 **사이클 존재**이므로 추가하려는 A→B 관계를 **거부**하거나 오류로 표시합니다. 이때 DFS 탐색은 **현재 추가하려는 간선을 제외한 기존 간선들로 구성된 그래프**에 대해 이루어져야 정확합니다. 코드 관점에서, 기존 관계 리스트를 순회하며 `if rel['type'] == 관계유형`인 것들만 뽑아 **인접 리스트(adjacency list)**를 구성한 뒤, 재귀 함수 혹은 스택을 이용해 탐색합니다 [83] [42]. 계획 문서의 `creates_cycle` 함수 구현이 이 논리를 잘 보여줍니다 [41] [42]. 특히, DFS 내에서 이미 방문한 노드를 추적하여 무한루프를 방지하고, 타겟 개념에서 시작해 소스 개념을 찾으면 True를 리턴하여 **사이클 발견 신호**를 보내는 구조입니다 [42].

구현 시 고려할 점은 **그래프의 크기**입니다. DFS는 최악의 경우 그래프 내 모든 노드를 방문하므로 O(V+E) 시간이 걸립니다. 하지만 교육용 개념 그래프는 일반적으로 수천 노드 이하 규모일 것이므로 현실적으로 큰 문제가 없을 것입니다. 그래도 최적화를 하자면, **관계 유형별로 분리된 그래프**를 관리하면 좋습니다. 즉, 선행관계용 adjacency 리스트, 포함관계용 리스트 등을 별도로 갖고 있으면 검사 범위가 줄어듭니다. 현재 `validate_relationship_logic`에서는 매번 주어진 관계유형에 대해 인접리스트를 만들어 DFS를 돌고 있습니다 [84]. 이를 개선한다면 QualityAgent가 처음 그래프를 받을 때 관계별 인접 구조를 만들어 두고, 각 관계 검사 시 재사용하도록 할 수 있습니다. 다만 그럴 경우 그래프 변경이 일어날 때마다 해당 구조를 업데이트해야 하므로, 일단은 간단히 **필요할 때마다 DFS**하는 방법이 이해하기 쉽고 버그 가능성도 낮습니다.

- **대칭성 검증:** `symmetric=True` 관계에 대해서는 **쌍방 간의 관계 존재 여부**를 확인해야 합니다. 구현은 비교적 단순합니다. 새로운 관계를 추가하거나 검증할 때, 그 관계의 타입이 대칭이라면 **그래프 내에 역방향 관계가 있는지** 검색합니다 [44] [85]. 계획 코드의 `find_relation` 함수는 주어진 그래프에서 특정 소스, 타겟, 타입을 가진 관계를 찾아 리턴하는 기능을 합니다 [86]. 이를 활용하거나, Python의 any/filter 등을 써서 `any(rel for rel in graph['relationships'] if rel['source_id']==target_id and rel['target_id']==source_id and rel['type']==type)` 같은 논리로 검사할 수 있습니다.
- 만약 역방향 관계가 **없으면**, 이는 대칭 관계가 완전히 맺어지지 않은 불완전 상태이므로 **경고(Warning)**를 기록합니다 [87]. 경고로 처리하는 이유는, 해당 역관계를 자동 추가할지 여부를 사람이나 후속 프로세스가 결정할 수 있도록 하기 위함입니다. 일부 경우 모델이 한쪽만 내놓았을 수 있으므로, Self-ImproverAgent가 이 경고를 보고 역관계를 추가 생성하도록 유도할 수도 있습니다. (예: "개념 X와 Y는 동등한 관계이나 한 방향밖에 정의되지 않았습니다. 반대 방향도 추가하세요."라는 프롬프트를 Self-Improver에 던져 자동 수정 가능).

- 만약 역방향 관계가 **존재하는데**, 해당 관계 유형이 사실 `anti_symmetric=True`로 정의된 타입이라면 이는 **논리 오류**입니다. 이런 경우는 QualityAgent에서 잡아내야 하는데, 현재 계획 코드에서는 anti_symmetric 속성을 직접 활용하는 검사는 구현되지 않았습니다. 이를 보완하기 위해 `validate_relationship_logic`에 다음과 같은 체크를 추가할 수 있습니다:

```
# Check 4: Anti-symmetry
if props.anti_symmetric and existing_graph:
    reverse_rel = find_relation(target_id, source_id, rel_type, existing_graph)
    if reverse_rel:
        errors.append(ValidationError(
            "ERROR",
```

```
                "antisymmetry_violation",
                f"{rel_type.value} is anti-symmetric, but reverse {target_id}→{source_id} also
exists",
                "Remove one of the conflicting relations or revise type if they are actually
identical concepts"
            ))
```

이 로직은 A→B (anti-symmetric인 관계)가 추가될 때 이미 B→A가 있으면 오류를 내는 형태입니다. 이렇게 하면 **잘못된 이중 정의**를 차단할 수 있습니다. (예: A 선행→B와 B 선행→A가 동시에 추가되는 실수를 방지)

- **코드 통합:** DFS 및 symmetry 검증은 `relationship_ontology.py`의 `validate_relationship_logic`에서 수행하는 것으로 앞서 설명했는데, 해당 함수를 QualityAgent가 호출하거나 RelationshipDefiner의 validate 단계에서 사용하게 됩니다 [51]. 코드를 작성할 때 `existing_graph`를 매번 전달해야 하며, RelationshipDefiner가 개별 관계 검증 시에는 self가 보유한 그래프를 넘기고, QualityAgent가 전체 검증 시에는 전체 그래프를 넘기는 형태로 일관되게 쓰면 되겠습니다. 또한 DFS `creates_cycle` 함수와 `find_relation` 함수는 `relationship_ontology.py` 내에 헬퍼로 구현하거나, QualityAgent 클래스의 staticmethod로 두어도 좋습니다. 계획안 코드에서는 ontology 모듈에 정의되었지만, QualityAgent에 넣으면 graph 데이터를 QualityAgent 내부에서 활용하기 쉬운 장점이 있습니다. 어느 쪽이든 큰 상관은 없으나, **ontology.py는 규칙 정의 중심, QualityAgent는 실행 중심**으로 구분짓는 편이 구조상 명확합니다.

- **검증 로직 검토:** DFS 및 대칭 검증을 추가함으로써 예상되는 상황을 한번 예로 들어보겠습니다. 만약 사용자 입력으로 개념 A와 B가 서로 선행 관계로 나오게 되어 Claude가 A→B와 B→A를 모두 관계로 제안했다고 합시다.

- RelationshipDefiner가 A→B (prerequisite) 출력 후 validateRelationship 실행 → `find_relation(B, A)`가 아직 그래프에 없으므로 이 순간에는 통과될 수 있습니다 (또는 symmetry 관련 anti_symmetric 체크에서 B→A 없는 것은 문제 안 삼음). A→B는 그래프에 추가.
- 이어서 B→A (prerequisite)도 출력되면 validateRelationship 실행 → 이번에는 그래프에 A→B가 있으므로, `anti_symmetric=True`인 prerequisite 속성에 의해 B→A 추가는 **antisymmetry_violation 오류**를 받을 것입니다. QualityAgent/RelationshipDefiner는 이 오류를 보고 B→A 관계를 **기각 또는 수정**하게 되겠죠.
- 최종적으로 A→B만 남고 B→A는 제외되어, **순환과 상호 선행 관계**가 제거됩니다.

- 만약 관계 유형이 `co-requisite`였다면 (대칭 필요), 첫 번째 A↔B 출력 시 B→A 없다고 **경고**가 나오고, 두 번째 출력에서 대칭 쌍이 채워지면 경고도 사라지는 흐름이 될 것입니다.

- **요약:** QualityAgent에 DFS 순환 탐지와 symmetry 검증을 넣는 것은 **그래프 정합성 유지의 핵심**입니다. 구현 난이도는 높지 않으나 (기본 그래프 탐색, 리스트 검색), 이를 **시기적절하게 호출**하는 것이 중요합니다. 또, DFS 함수 등은 재사용 가능하므로 모듈 전역 함수로 빼거나, QualityAgent 내부에서 캐싱해도 됩니다. 결과적으로, 이러한 검증 로직을 통해 시스템은 **순환 없는 계층 구조**와 **요구되는 대칭 관계의 쌍**을 보장받게 됩니다. (연구 보고서에서도 이러한 검증 규칙 준수를 강조하고 있습니다 [5].)

5. ExternalValidatorAgent 신설: 외부 SPARQL 조회 및 내부 매핑 로직 설계

시스템 내부 지식만으로 검증하는 QualityAgent와 별개로, **외부의 지식베이스**를 활용하여 관계의 진위를 검증하거나 새로운 연관 정보를 얻는 **ExternalValidatorAgent**를 도입하는 방안을 살펴봅니다. 이는 예컨대 Wikidata나

DBpedia같은 오픈 지식 그래프, 혹은 수학 교육용 온톨로지(OntoMathEdu, Palantir Foundry Ontology 등)를 조회하여 Claude가 제안한 관계를 교차검증하는 역할을 할 수 있습니다 [88].

- **역할 정의:** ExternalValidatorAgent는 **(개념 A, 개념 B, 관계 타입)**을 입력으로 받아, **외부 지식 소스에서 관련 정보를 질의**하고 그 결과를 토대로 **내부 관계의 유효성**을 평가하거나 **추가 정보**를 부여합니다. 두 가지 주요 기능 시나리오가 있습니다:
- **관계 검증:** Claude가 "A는 B의 특수한 경우이다 (extension 관계)"라고 판정했을 때, 외부 지식베이스에 A와 B의 관계를 물어봅니다. 만약 외부에도 유사한 관계 진술이 있으면 신뢰도를 높여주고, 정반대 관계가 있으면 오류를 지적하거나 낮은 신뢰도로 표시하는 식입니다.

- **관계 제안:** Claude가 관계를 **모호하게 표현**했거나 `"alternative_classifications"`로 후보만 준 경우, 외부 KB를 조회하여 그 둘에 어떤 관계로 연결되어 있는지 찾아 **관계 추천**을 할 수 있습니다. 이는 새로운 관계 유형 발굴에도 도움을 줄 수 있습니다 (예: 외부에는 A와 B가 "연관개념(related)"으로만 표시돼 있다면, 내부 taxonomy에 없는 느슨한 관계라는 신호로서 "supports" 등의 새 관계를 고려).

- **외부 소스 및 SPARQL 설계:** 외부 지식은 **구조화된 RDF 데이터**가 있는 소스를 가정하겠습니다 (Wikidata, DBpedia 등은 SPARQL 엔드포인트로 질의 가능). ExternalValidatorAgent는 내부 개념을 외부 리소스와 연결하는 **매핑 작업**이 선행되어야 합니다. 이를 구현하는 단계별 설계는 다음과 같습니다:

- **개념 식별자 매핑:** 내부 `concept_id` 혹은 이름을 외부 KB의 식별자로 변환합니다. 간단한 방법은 개념 이름으로 외부에서 **레이블 검색**하는 것입니다. 예를 들어 DBpedia의 경우 `SELECT ?s WHERE { ?s rdfs:label "Concept Name"@en }` 같은 쿼리로 URI를 찾을 수 있습니다. 혹은 미리 내부-외부 URI 매핑 테이블을 구축해둘 수도 있습니다 (예: concept "Derivative" → Wikidata Q43122). 이 단계는 모호성을 처리해야 하므로, ExternalValidatorAgent 내에 **개념 검색/매핑 함수**를 둡니다. 실패하거나 다의어가 나오면 인간 도움이나 우선순위 규칙이 필요할 수 있습니다.

- **SPARQL 쿼리 생성:** 매핑된 외부 리소스들에 대해, **관계 존재 여부**를 묻는 SPARQL 쿼리를 생성합니다. 예를 들어 내부에서 "prerequisite" 관계라면 외부 Ontology에는 없을 가능성이 높지만, "related to"나 "see also" 등 간접 관계를 탐색할 수 있습니다. 쿼리 예시 (DBpedia 가정):

```
SELECT ?pred ?objLabel WHERE {
  ?sub rdfs:label "ConceptA"@en .
  ?obj rdfs:label "ConceptB"@en .
  ?sub ?pred ?obj .
  FILTER(
    ?pred IN (dbc:Prerequisite, dbp:requirement, skos:related)
  )
  SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
}
```

이 쿼리는 ConceptA와 ConceptB에 해당하는 리소스를 찾고, 둘 사이의 관계(pred)가 미리 정의한 몇 가지 후보 (예: DBpedia 커스텀 `Prerequisite` 프라퍼티나 `skos:related` 등) 중에 있는지 검색합니다. 실제로 DBpedia에 그런 프라퍼티가 없으면 결과가 없을 것이고, Wikidata라면 교육 개념의 선행지식 속성이 있을 수도 있으니 바꿔 질의하는 등 시도합니다.

- **결과 해석:** SPARQL 결과로 (A -[pred]-> B) 관계들이 나오면, 그 pred를 내부 taxonomy의 관계와 매핑해볼 수 있습니다. 예컨대 Wikidata에서 "has part"로 연결돼 있으면 내부에서는 "component-of" 관계에 해당할 수 있습니다. ExternalValidatorAgent는 **사전 정의된 관계 매핑표**를 이용해 외부 프레디켓을 내부 관계 유형으로 대응시켜야 합니다. 이 매핑표는 수동으로 작성하거나, OntoMathEdu처럼 도메인 전문 온톨로지와 우리

시스템 taxonomy의 교차표가 있다면 활용합니다. 연구 자료에서 OntoMathEdu, Math-KG 등을 비교한 것도 이러한 매핑을 위한 기초 자료일 것입니다 [88] .

- **판단 및 반환:** 외부 정보와 내부 추론 결과를 비교하여, **검증 상태**를 판단합니다. 예를 들어 내부에서는 "prerequisite"로 판단했는데 외부에는 아무 연관이 없다면 “검증 불가” 정도로 둘 수 있고, 외부에 오히려 B가 A의 상위개념(superclass)이라고 나와 있으면 “모순: A is broader concept than B externally” 같은 메시지를 줄 수 있습니다. 최종적으로 ExternalValidatorAgent는 `{"verification": "confirmed"}` 또는 `{"verification": "contradicted", "evidence": "..."}`, 혹은 신뢰도 점수를 조정한 새 결과 등을 반환할 수 있을 것입니다. 이 반환 형식은 QualityAgent와는 별도로 정의하고, MetaOrchestrator에서 **선택적으로 이 결과를 참고하여 품질 게이트 통과 여부에 반영**할 수 있습니다.

- **내부 구조 연계:** ExternalValidatorAgent를 새로 추가한다면, Orchestrator의 상태 흐름에서 **QualityCheck 이후**나 **QualityCheck 일부로** 실행할 수 있습니다. 두 가지 통합 방식이 있습니다:

- QualityAgent 내부에서 ExternalValidator를 호출하여 **semantic check**의 연장선으로 활용. 예를 들어 QualityAgent의 `_validate_semantics` 단계에서 각 관계에 대해 ExternalValidatorAgent.query(conceptA, conceptB, type)을 호출하고, 결과에 따라 경고/오류를 추가할 수 있습니다. 하지만 외부 지식은 한정적일 수 있으므로, 없는 정보를 가지고 오류로 판단하면 False positive가 생길 위험이 있습니다. 그래서 **강제 오류보다는 참고용 경고** 위주로 활용하는 것이 안전합니다. (e.g., "External KB found no link between A and B - verify domain context" 정도의 경고)
- Orchestrator 상에서 QualityAgent 검증 후, **추가적인 validation step**으로 ExternalValidatorAgent를 호출. 이 경우 QualityAgent는 내부 검증만으로 Passed 처리하고, ExternalValidator에서 모순이 발견되면 별도의 HITL 또는 Socratic 피드백을 트리거하는 방식입니다.

- **구현 및 의존성:** ExternalValidatorAgent는 Python에서 **SPARQL 쿼리**를 실행할 수 있어야 하므로, `SPARQLWrapper` 라이브러리 등을 사용하거나 HTTP 요청을 날릴 수 있어야 합니다. 이는 **외부 인터넷 의존성**을 뜻하므로, 시스템 운용 환경에서 보안/속도 이슈를 고려해야 합니다. 만약 인터넷 접속이 안 되는 환경이라면, 주요한 외부 지식 그래프를 **로컬에 캐시**하거나, 관계 검증에 필요한 최소한의 트리플 셋을 내장 데이터로 갖고 있어야 할 것입니다 (예: “중학교 수학 개념 X와 Y는 교육과정 상 연계되어 있다” 같은 지식을 별도 yaml에 저장).

ExternalValidatorAgent의 **매핑 로직**(내부 개념→외부 URI, 외부 프레디켓→내부 관계 유형)은 시간이 드는 작업이지만, **재사용성**을 높이기 위해 별도 모듈이나 설정 파일로 관리하는 것이 좋습니다. 향후 새로운 외부 데이터 소스를 연결할 때도 이 매핑만 추가하면 되도록 **확장성**을 고려해야 합니다.

- **예시 시나리오:** 예를 들어 내부 개념 "삼각형의 둘레(perimeter of triangle)"와 "삼각형" 사이에 Claude가 "prerequisite" 관계를 제안했다고 합시다. ExternalValidatorAgent는 Wikidata에서 "삼각형의 둘레" 항목(QXXX)을 찾아보고, 그 항목의 속성 중 "has part"나 "aspect of" 등 삼각형(QYYY)를 가리키는 것을 발견했다고 합시다. 만약 "aspect of 삼각형"이라는 트리플이 있으면, 이는 "perimeter of triangle **is an aspect of** triangle"이라는 뜻이므로, 우리 내부의 "extension" 관계 (더 일반 개념의 구체적 사례)와 비슷할 수 있습니다. 그런데 Claude는 prerequisite으로 분류했으니, ExternalValidator는 "External data suggests 'perimeter of triangle' is a part of 'triangle' rather than a prerequisite knowledge."라는 피드백을 줄 수 있습니다. 이를 받으면 Self-ImproverAgent는 관계 재분류를 시도하거나, 최소한 경고로 표기해두고 넘어갈 수 있습니다.

- **효과:** ExternalValidatorAgent를 잘 활용하면 **모델의 한계를 보완**하여, Claude가 놓칠 수 있는 사실적 검증을 수행할 수 있습니다. 특히 잘못된 관계를 걸러내 정확도를 높이고, 새로운 관계를 학습하거나 **외부 용어**(시소러스 관계 등)를 내부 지식에 흡수하는데 도움이 됩니다. 연구 단계에서는 시간상 제외됐던 부분이겠지만, Phase

3나 향후 Phase 4 개선으로 고려할만한 가치가 있습니다. 다만 앞서 언급한 **의존성 문제와 커버리지 한계**를 인지하고, 이 기능은 **옵션** 또는 **보조적인 수단**으로 두는 것이 안전합니다.

6. 잠재적 병목, 비용 이슈, 외부 의존성 분석 및 보완책

마지막으로, 위 개선사항들을 모두 구현했을 때 시스템에 발생할 수 있는 **성능 상의 병목**, **비용 증가**, **타 시스템 의존성** 등의 문제점을 정리하고, 이에 대한 보완 대책을 제시합니다.

- **(성능) 다단계 에이전트 호출 증가:** 개선안 적용으로 에이전트 간 왕복(call)이 늘어나고, QualityAgent 검사, ExternalValidator 질의 등 추가 단계가 생깁니다. 이는 곧 **응답 지연 증가**로 이어질 수 있습니다. 특히 CoT 도입(개선 2)으로 Claude의 1회 응답이 길어지고, QualityAgent의 그래프 검증은 관계 수에 비례하여 시간 소모가 있습니다. **보완책:** 병렬화(개선 6)를 적극 활용하여 병목을 줄이고, **캐싱**을 도입합니다. 예를 들어 이전에 분석한 적 있는 개념 쌍에 대한 관계는 캐시해두고 같은 질의가 오면 바로 결과를 재활용하거나, ExternalValidator의 외부 질의 결과도 일정 시간 저장해두어 반복 쿼리를 피합니다. 또한 필요할 때만 상세 검증을 하는 **조건부 검사** 전략을 사용합니다. (예: confidence 높고 Low risk한 변화엔 QualityAgent의 full-check 생략, 불확실성 높을 때만 시행 등).

- **(비용) LLM 토큰 사용량 및 API 호출 비용:** CoT 프롬프트와 추가 출력으로 **토큰 수**가 증가하고, 병렬화시 여러 호출 동시 수행으로 **API 사용량**이 늘어날 수 있습니다. 또한 QualityAgent나 ExternalValidatorAgent는 그래도 비교적 가벼운 편이지만, Self-ImproverAgent 등의 재시도가 늘면 LLM 호출 횟수 자체가 증가할 것입니다. **보완책:** 우선 프롬프트 설계를 최적화하여 **불필요한 수다는 줄이고 핵심 지시만 담도록** 해서 토큰 낭비를 막습니다. 예를 들어 CoT 단계에서도 필요한 질문 이외에는 넣지 않는 식입니다. 그리고 **기존 루프의 반복 횟수를 줄이는** 것이 비용 절감의 핵심인데, 다행히 개선 1,2 등을 통해 루프 반복(재진단)이 감소할 것으로 기대됩니다[7]. 따라서 초기에는 약간 토큰이 늘어도, **overall iteration count 감소로 비용은 오히려 줄 수도** 있습니다. 실제 효과는 모니터링을 통해 확인하고, 필요시 **매 단계 비용 산정**을 해가며 과도하게 비싼 부분을 손볼 수 있습니다 (예: ExternalValidator의 외부 API 요금 등이 있다면 호출 빈도를 조절).

- **(동시성) 병렬 처리에 따른 동기화 문제:** 개선 6을 적용하여 여러 스레드/태스크가 병렬로 LLM 호출이나 데이터를 다룰 경우, 예기치 못한 **race condition**이나 **상태 불일치** 문제가 발생할 수 있습니다. 예를 들어 두 개의 sub-task가 동일한 전역 변수에 기록할 때 충돌하거나, 결과 병합 시점을 놓쳐 잘못된 결론을 낼 위험입니다. **보완책:** 병렬 수행 영역을 최대한 서로 **독립적인 stateless 함수**로 구성합니다. 필요한 경우 스레드 세이프한 큐나 잠금(lock)을 사용하되, 범위를 최소화하여 lock contention을 줄입니다. Python의 GIL이 일부 serialization을 유발할 수 있으므로, CPU 연산이 많은 부분은 병렬화 대상에서 제외하고 I/O 위주 호출을 병렬화하는 방향으로 합니다. 또한 멀티스레드 디버깅이 어려우므로, **철저한 로그**를 남겨 동작 순서를 추적할 수 있게 하고, 단위 테스트에서는 병렬 부분을 직렬로도 실행해 결과 일관성을 검증해봅니다.

- **(HITL에 의한 지연)**: Human-in-the-loop 단계는 본질적으로 **사람의 반응 시간을 기다려야** 하므로 시스템 흐름이 오래 중단될 수 있습니다. 이는 실시간 상호작용 시스템에는 부적합할 수 있습니다. **보완책:** HITL 단계를 **비동기 이벤트**로 다루어, 메인 루프는 새로운 세션을 받아 처리하거나 다른 작업을 진행할 수 있도록 합니다. 예를 들어 변경안을 저장하고 사용자에게 알린 뒤, Loop는 해당 세션을 일시 중단하고 다른 요청을 처리하다가, 사용자가 승인 신호를 보내면 그때 다시 이어서 마무리하는 구조입니다. 이렇게 하면 전체 스루풋은 유지할 수 있습니다. 또 하나는 **HITL 사용 여부를 설정**으로 두어, 배포 환경에 따라 끌 수 있게 하는 것입니다. 개발/테스트 시에는 HITL를 끄고 자동 진행해서 속도를 높이고, 프로덕션에서는 켜는 식입니다.

- **(외부 의존성 및 신뢰성):** ExternalValidatorAgent처럼 외부 API나 DB에 의존하면, **네트워크 지연**이나 **엔드포인트 장애**가 시스템에 영향을 줄 수 있습니다. 또한 외부 지식의 품질이 담보되지 않으면 오히려 잘못된 피드백을 줄 위험도 있습니다. **보완책:** 외부 의존 모듈에는 **타임아웃**과 **예외 처리**를 철저히 해 두고, 실패 시 해당 검증을 건너뛰거나 기본값으로 처리하도록 합니다. 예를 들어 SPARQL 쿼리가 5초 내 응답 없으면 "외부 검증 생략" 경고만 주고 넘긴다든지 합니다. 또한 외부 지식과 내부 추론이 충돌할 경우, **내부 결과를 완전히 폐기하지 말고 사람 검토 대상**으로 돌리는 보수적 접근이 필요합니다. (즉, ExternalValidator는 권고/경고 역할, 최종 판

단은 Human 또는 내부 알고리즘이 하도록). 그리고 추후 외부 소스가 바뀔 수 있으므로, **버전 관리**와 **캐싱 전략**을 세워서 결과의 재현성을 확보해야 합니다.

- **(복잡도 증가로 인한 버그 가능성):** 여러 개선을 한 번에 도입하면 시스템이 복잡해져 **예측하지 못한 상호작용 버그**가 생길 수 있습니다. 예컨대 QualityAgent가 잡아낸 이슈를 Self-Improver가 고치다가 다른 부분을 망가뜨리는 등 루프 간섭 문제가 있을 수 있습니다. **대응책: 단계적 구현 및 테스트**를 강력 권장합니다. 실제 계획(Roadmap)에서도 3개의 Phase로 8주에 걸쳐 순차 적용하도록 되어 있는데 [89] [90], 이 순서를 존중하여 한 번에 한두 개 개선씩 넣고 검증하는 게 좋습니다. 구체적으로, Phase 1의 불확실성 모델링과 CoT를 먼저 적용한 뒤 충분히 테스트 (Phase 1 validation)하고 [91], Phase 2의 동적 게이트와 온톨로지/QualityAgent를 적용, 검증, 마지막으로 Phase 3의 HITL과 병렬화를 적용하는 식입니다. 이렇게 하면 문제가 발생해도 어느 부분에서 기인했는지 좁혀서 찾을 수 있고, 롤백도 용이합니다 [92].

- **(타 시스템 연계):** Claude 모델 자체의 버전 업이나, 다른 LLM으로 교체, 혹은 UI 시스템과 연계 등 **외부 연동 사항**이 향후에 있을 수 있습니다. 우리의 개선들이 Claude 모델에 의존적인 부분 (프롬프트 설계 등)이 많기 때문에, 만약 API나 모델이 변경되면 조정을 해야 합니다. 예를 들어 Claude의 다음 버전이 CoT 없이도 추론 잘 한다면 체인을 줄일 수도 있고, 반대로 모델이 바뀌어 prompt length 한도가 달라지면 조절해야 합니다. **대응책:** 이러한 변경 가능성에 대비해 **설계 유연성**을 유지합니다. 프롬프트 내용들은 하드코딩하기보다 별도 설정파일이나 템플릿으로 두어 쉽게 수정 가능하게 하고, magic number나 임계값(예: 0.85, 20, 0.80 등)은 `config.py`로 관리하여 조정성을 높입니다. 또 Documentation를 업데이트해 두어, 새로운 팀원이 오더라도 왜 이런 설계를 했고 어느 부분을 손대면 되는지 이해할 수 있게 합니다.

종합 의견: 전체적으로, 제안된 개선사항들은 시스템의 **지능적 판단 능력**과 **신뢰성**을 대폭 향상시킬 것으로 기대됩니다. 다만 그 대가로 **구조의 복잡성 증가**와 **자원 소모**가 약간 늘어나는 것은 피할 수 없습니다. 이를 관리하기 위해서는 상기 언급한 방법들처럼 **병렬 처리로 성능을 보완**하고, **캐싱/조건부 실행으로 불필요한 낭비를 줄이며**, **단계별 적용과 테스트**로 안정성을 확보하는 전략이 필요합니다. 특히 사람의 전문 지식을 투입하고 외부 지식을 참조하는 방향은, 단기적으로는 속도를 늦출지 몰라도 장기적으로 **도메인 적합도와 오류 방지** 측면에서 큰 이득이 있을 것입니다.

결론적으로, Claude 기반 자가개선 루프 시스템은 이 계획을 통해 **초기 설정된 목표 지표**(예: 정확도 +15%, 논리적 일관성 위배 0건 등 [93])에 근접하거나 초과할 것으로 보입니다. 다만 구현 과정에서 예상치 못한 이슈도 분명 발생할 것이므로, 유연한 대처와 지속적인 모니터링이 중요합니다. 이 보고서에서 다룬 코드 수준의 세부사항과 권고안을 참고하여 개발을 진행한다면, Phase 1~3의 개선을 성공적으로 마무리하고 한층 고도화된 **자가개선 AI 교육 도우미 시스템**을 구축할 수 있을 것입니다. [94] [95]

---

[1] [2] [3] [4] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30]
[31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59]
[60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [78] [79] [80] [82] [83] [84] [85] [86] [87] [89] [90] [91] [92] [93] [94] [95]
CODE-IMPROVEMENT-PLAN.md
file://file_00000000d46c61f5bd1b076bc096f918

[5] [88] Claude Sonnet 4.5 기반 개념 관계 정의 체계 개선 연구 보고서.pdf
file://file_000000003bac61fd9f40654a483d9998

[73] [74] [75] [76] [77] LangGraph State Machines: Managing Complex Agent Task Flows in Production - DEV Community
https://dev.to/jamesli/langgraph-state-machines-managing-complex-agent-task-flows-in-production-36f4

[81] State Transitions Not working as Expected in Agent Setup #3343
https://github.com/langchain-ai/langgraph/issues/3343