ChatGPT

P1-2 파일 시스템 에이전트와 P2-SU 자기 업데이트 엔진 통합 작업 지시서

시스템 연동 개요

P1-2 **파일 시스템 에이전트**(`file_agent.py`)는 플러그인 기반의 AST 리팩토링 엔진으로서, CLI 명령 `invoke refactor`를 통해 다양한 코드 수정 규칙을 실행합니다 [1]. P2-SU **자가 업데이트 엔진**(`proposer.py`)은 시스템 상태를 지속적으로 모니터링하여 업데이트가 필요한 영역을 자동으로 감지하고 개선안을 제안·적용하는 모듈입니다. 예를 들어, **pip 패키지의 신규 버전**이나 **코드의 DeprecationWarning** 발생을 탐지해 업데이트 기회를 식별하고, 이를 바탕으로 수정 제안서를 생성합니다 [2].

두 시스템의 통합 목표는 **자가 개선 루프**를 구축하는 것입니다. 즉, `proposer.py`가 발견한 개선 사항(예: **의존성 버전 상승**, **사용 중단 예정 API 교체**, **코드 스타일 정리** 등)을 자동으로 P1-2의 리팩토링 규칙에 매핑해 실행하는 것입니다. `proposer.py`는 감지된 업데이트 필요 사항마다 해당되는 리팩토링 **규칙**을 결정하고, 그에 맞는 `invoke refactor` 명령을 구성하여 **파일 에이전트**에 전달합니다. 예를 들어 `proposer.py`가 어떤 패키지 버전이 오래되었다고 판단하면 "의존성 업데이트" 규칙으로, 특정 함수가 Deprecated되었다면 "API 교체" 규칙으로 매핑하는 식입니다. 파일 에이전트는 전달받은 규칙 명을 기반으로 플러그인 모듈을 찾아 AST 변환을 수행하며 [1], 그 결과로 코드 수정 사항(diff)을 생성하거나 파일에 적용합니다.

이 연동 구조는 **확장성**을 최우선으로 고려합니다. P1-2 에이전트가 **규칙 플러그인 아키텍처**로 구현되어 있어 새로운 리팩토링 규칙을 쉽게 추가할 수 있는 만큼, P2-SU 엔진도 유연하게 대응해야 합니다. `proposer.py`는 **규칙 레지스트리**나 정책 파일을 참고하여 현재 지원하는 모든 자동화 규칙을 인지하고, 각 규칙에 해당하는 **감지 로직**을 모듈화할 것입니다. 새로운 자동 개선 시나리오가 생기면, 단지 새로운 규칙 플러그인 파일을 `rules/` 디렉터리에 추가하고 [3], `proposer.py` 쪽에서 그 규칙에 대한 감지 함수를 추가하는 것만으로 통합이 이뤄집니다. 이로써 파일 에이전트 측 코어 로직 수정 없이도(개방-폐쇄 원칙 준수) Self-Update 기능을 확장할 수 있습니다. 또한, P2-SU 엔진은 **SELF_UPDATE_POLICY.md**와 같은 정책을 참조하여 어떤 개선안을 자동 적용할지 vs. 수동 검토로 남길지를 결정함으로써, **안전성과 자동화 수준의 균형**을 맞춥니다.

요약하면, **자가 업데이트 엔진(P2-SU)**이 개선 필요 항목을 **감지**하고 -> 해당 **리팩토링 규칙(P1-2)**을 식별해 -> `invoke refactor` 명령으로 **실행**하며 -> 결과를 검증 및 반영하는 흐름으로 두 시스템이 유기적으로 연동됩니다.

Refactor 명령 생성 포맷 정의

P2-SU 엔진이 `invoke refactor` 명령을 생성할 때는, **파일 에이전트 CLI의 인자 규격**에 맞춰야 합니다. 기본 형태는 다음과 같습니다:

- `invoke refactor --file <대상파일> --rule <규칙명> [추가 옵션들]`

여기서 `<대상파일>`은 수정할 파일 경로이고 `<규칙명>`은 적용할 리팩토링 규칙(plug-in 모듈 이름)입니다. 추가 옵션으로는 **미리보기용** `--dry-run` 플래그와 **자동 적용용** `--yes` 플래그가 중요합니다. `--dry-run`을 지정하면 실제 파일을 수정하지 않고 변경 사항(diff)만 출력하며 [4], `--yes`를 지정하면 사용자 확인 없이 바로 파일에 변경을 적용합니다 [5]. (`--yes` 플래그가 없을 경우 파일 에이전트는 안전장치로 수정 사항을 실제로 커밋하지 않습니다 [6].)

또한 `invoke refactor`에는 기타 부가 기능으로 `--list-rules` (사용 가능한 규칙 목록 조회)와 `--explain <규칙>` (특정 규칙의 설명 출력)이 있지만 [7], 이는 주로 사용자 상호작용용이므로 자동화 엔진이 명령 생성 시에는 보통 활용하지 않습니다.

규칙별 추가 인자 전달: Self-Update 엔진이 특정 규칙을 실행할 때 부가 정보가 필요한 경우, 명령에 규칙별 파라미터를 포함시켜야 합니다. 현재 설계된 CLI에서는 `--file`과 `--rule` 이외의 임의 인자를 직접 받도록 되어 있지 않지만, **확장 설계**로 각 규칙에서 `**kwargs`를 받을 수 있으므로 파일 에이전트 측에서 특정 플래그를 해석하도록 구현할 수 있습니다. 예를 들어: - **requirements.txt 업데이트**: 특정 패키지 버전을 올리는 규칙(`update_dependency`)의 경우, `--package`와 `--version` 같은 커스텀 인자를 지원하도록 하여, Self-Update 엔진이 어떤 패키지를 어떤 버전으로 올릴지 명령에 포함시킬 수 있습니다. 예시: `invoke refactor --file=requirements.txt --rule=update_dependency --package Flask --version 2.3.0 --dry-run` - **Deprecated API 교체**: Deprecated된 함수나 클래스를 새 API로 변경하는 규칙(`replace_api`)을 가정하면, `--old-name`과 `--new-name` 인자를 활용해 교체 대상 식별자를 전달할 수 있습니다. 예시: `invoke refactor --file=myapp/util.py --rule=replace_api --old-name OldFunc --new-name NewFunc --dry-run` - **Lint 수정 적용**: 코드 린트 오류를 일괄 수정하는 규칙(`lint_fix`)의 경우 특별한 추가 인자 없이 대상 파일만 지정하면 됩니다. 예시: `invoke refactor --file=src/main.py --rule=lint_fix --yes`

위와 같은 형식으로 `proposer.py`는 상황에 따라 `--dry-run`과 `--yes`를 조합해 사용합니다. **권장 시나리오:** 자동 업데이트 엔진이 우선 `--dry-run`으로 변경 사항을 산출하여 로그나 보고용으로 활용하고, 자동 승인 조건에 해당하면 `--yes`로 재실행하여 실제 반영합니다. 예컨대 “requirements.txt에 Flask 패키지 2.3.0으로 업그레이드” 건이 발견되면: 1) `invoke refactor --file=requirements.txt --rule=update_dependency --package Flask --version 2.3.0 --dry-run`으로 diff를 생성 및 검토하고, 2) 정책상 자동 적용 대상이면 `--yes`를 붙여 최종 적용하는 흐름입니다.

SELF_UPDATE_POLICY.md 초안 제안

SELF_UPDATE_POLICY.md 파일은 자동 업데이트 대상 규칙들과 승인 정책을 정형화하여 기록해 두는 **버전 관리 정책 문서**입니다. Self-Update 엔진은 이 파일을 참조해 각 리팩토링 규칙의 속성(위험도, 자동 승인 여부, 테스트 필요 여부 등)을 판단하고, 그에 따라 동작을 결정합니다. 정책을 Markdown 문서로 둠으로써, 팀원들이 쉽게 열람·편집하고 변경 이력을 추적할 수 있습니다.

아래는 **SELF_UPDATE_POLICY.md**의 초기 초안 예시입니다:

```
# Self-Update Engine Policy (Draft)

각 자동 업데이트 규칙에 대한 메타데이터와 승인 정책을 정의합니다.

| **Rule ID**          | **Category (분류)** | **Risk Level** | **Auto Approve** | **Test Required** | **Description (설명)**                          |
|----------------------|---------------------|----------------|------------------|-------------------|-------------------------------------------------|
| **update_dependency** | 의존성 관리         | 낮음 (Low)     | 예 (Yes)         | 예 (Yes)          | 패키지의 패치/마이너 버전을 최신으로 업그레이드. |
| **replace_deprecated** | 코드 유지보수     | 중간 (Medium)  | 예 (Yes)         | 예 (Yes)          | 사용 중단 예정(API Deprecation) 항목을 권장 대안으로 교체. |
| **lint_fix**         | 코드 스타일         | 낮음 (Low)     | 예 (Yes)         | 아니요 (No)       | 코드 포매팅 및 린트 오류 수정 (논리 변화 없음). |
| **add_docstrings**   | 문서화              | 낮음 (Low)     | 예 (Yes)         | 아니요
```

```
(No)        | Docstring이 없는 함수에 템플릿 주석 추가.       |
```

(위 표는 초기 제안이며, 프로젝트 상황에 맞게 항목과 값은 조정될 수 있습니다.)

정책 파일의 각 컬럼 의미: - **Rule ID**: 파일 에이전트의 규칙 이름(identifier)입니다. Self-Update 엔진은 이 이름으로 규칙을 식별하고 `invoke refactor --rule` 인자로 사용합니다. - **Category**: 해당 규칙의 유형이나 범주를 나타냅니다 (예: 의존성, 코드 품질, 보안 등). - **Risk Level**: 업데이트의 위험도 평가입니다. 팀 합의에 따라 낮음/중간/높음 등급으로 분류하며, 추후 자동 승인 여부 결정에 참고합니다. - **Auto Approve**: 자동 승인이 가능한지 여부입니다. `True/False` 또는 "예/아니요"로 표시하며, True인 경우 검증 절차를 통과하면 사용자의 추가 확인 없이 변경을 적용합니다. - **Test Required**: 해당 규칙 적용 후 **테스트**를 필수로 거쳐야 하는지 여부입니다. True인 경우, 변경 적용 후 자동으로 `pytest` 등을 실행해 이상이 없을 때만 최종 승인합니다. - **Description**: 규칙의 간략한 설명으로, 어떤 변경을 하는지 서술합니다.

이 정책 문서는 **규칙별로 한 줄씩** 갱신되며, 새로운 self-update 리팩토링 규칙을 도입할 때마다 해당 메타정보를 추가해야 합니다. 예를 들어 `rename_variable`이라는 새로운 규칙을 만들어 Self-Update에 활용한다면, Rule ID와 분류, 위험도 등을 평가하여 이 파일에 한 줄을 추가합니다. 이렇게 하면 `proposer.py`가 실행될 때 최신 정책을 읽어들여, 각 규칙을 올바른 방식으로 처리할 수 있습니다.

자동 승인 기준 설계

자동 코드 수정이라 해도 모든 변경을 무조건 즉시 적용하는 것은 위험할 수 있으므로, **어떤 경우에 자동 승인(자동 적용)할지에 대한 명확한 기준**을 세워야 합니다. 아래는 자동 승인을 판단하는 주요 기준과 원칙입니다:

- **모든 자동 적용은 테스트 검증 필수**: 자동 승인이 이루어지기 전에 반드시 전체 테스트 스위트(예: `pytest`)를 실행하여 변경에 따른 **회귀가 없는지 확인**해야 합니다. Self-Update 엔진은 `Test Required`가 True로 지정된 규칙에 대해서, 리팩토링 적용 직후 테스트를 구동하고 **exit code=0(성공)**인 경우에만 변경 사항을 확정합니다. 테스트 실패 시 해당 변경은 자동 승인을 취소하고 롤백합니다 (자세한 프로세스는 아래 흐름에서 설명).

- **낮은 위험도의 리팩토링**: 사소한 변경이나 형식적인 개선은 자동 승인 대상입니다. 예를 들어 **코드 스타일 정정** (PEP8 준수, 불필요한 공백 제거 등)이나 **주석/Docstring 추가**와 같은 변경은 비즈니스 로직에 영향이 없으므로 위험도가 낮습니다. 이러한 규칙들은 대체로 `Risk Level: 낮음`으로 분류되며, **Auto Approve = True**로 설정합니다. 테스트 역시 통과 가능성이 높으므로 `Test Required`는 선택적입니다 (포매팅이나 주석 변경만 있는 경우 테스트를 굳이 돌리지 않아도 무방하나, 시스템 일관성을 위해 가급적 돌려보는 것이 좋습니다).

- **의존성 패치 버전 업그레이드**: 외부 라이브러리의 **패치 버전 혹은 호환되는 마이너 버전** 업은 일반적으로 하위 호환성이 유지되므로 낮은 위험으로 간주됩니다. 예를 들어 `requests 2.28.1 -> 2.28.2`나 `Flask 2.2 -> 2.3`와 같은 변경입니다. 이러한 업데이트는 **자동 승인 가능**하며, 단 `requirements.txt` 수정 후 **모든 테스트가 통과**해야 최종 확정합니다. 패치/마이너 업데이트는 보통 버그 수정과 개선 사항이므로, 최신 상태 유지를 위해 자동화된 적용이 권장됩니다. 단, 데이터베이스 마이그레이션 등이 수반되지 않는 한에서입니다.

- **Deprecated API 교체**: 오래된 함수나 클래스 등을 새로운 대체재로 변경하는 리팩토링은 중간 정도의 위험으로 볼 수 있습니다. 변경 범위가 넓을 수 있지만 대부분 패턴이 기계적으로 치환되므로, **테스트가 녹색(Green)이라면 자동 머지**할 수 있습니다. 예를 들어 Python 3.x에서 deprecated된 표준 라이브러리 함수를 새로운 것으로 대체하거나, 특정 라이브러리의 API 변경을 따라가는 수정은, **테스트 통과를 전제로 자동 적용**합니다. 이때 위험도 평가를 중간으로 한 이유는, 테스트에서 잡지 못하는 미묘한 변화(예: 동작은 되지만 성능이나 경고

측면에서의 영향)가 있을 수 있기 때문입니다. **따라서 해당 변경에 대한 테스트 커버리지가 충분히 있는지**를 사전에 판단하고, 필요하다면 `Test Required = True`로 설정하여 반드시 회귀 테스트를 거치도록 합니다.

- **높은 위험도의 변경**: **주요 버전 업그레이드**(Breaking Changes 가능)나 **복잡한 코드 리팩토링**은 자동 적용해서는 안 됩니다. 예를 들어 Django 3 -> 4로 올리는 경우나, 함수의 비동기화와 같이 로직을 크게 변경하는 리팩토링은 `Risk Level: 높음`으로 분류하고 **Auto Approve = False**로 정책에 명시합니다. 이러한 변경 사항은 Self-Update 엔진이 감지하더라도, **자동으로 PR이나 diff만 생성하고 사람 검토를 요구**하게 설계합니다. 즉, 시스템이 "제안서"까지만 준비하고 최종 승인은 개발자가 수동으로 하도록 유도합니다. 이는 잠재적인 사이드 이펙트를 인간이 한 번 더 점검함으로써 안정성을 확보하기 위함입니다.

- **자동 승인 예외 기준**: 일반적으로 낮음~중간 위험 + 테스트 통과이면 자동 승인을 허용하고, 높은 위험 또는 테스트 실패이면 자동 승인을 보류/취소합니다. 추가로, **테스트 커버리지 부족**이나 **변경 규모가 너무 큰 경우**도 자동 적용을 피하도록 보수적으로 운영합니다. 예를 들어 변경이 프로젝트 여러 모듈에 광범위하게 영향하면, 설령 각 부분의 위험도는 낮아도 전체적으로는 높아질 수 있으므로 인간 검토를 받는 것이 좋습니다. 이러한 세부 기준들은 SELF_UPDATE_POLICY.md의 각 규칙 메타데이터 (`risk_level`, `test_required`)와 별도로, Self-Update 엔진 내부 로직에 검증 루틴을 넣어 구현합니다.

요약하면, **"작은 변화 + 검증 완료"는 자동 적용**, **"큰 변화 또는 불확실성 존재"는 개발자 확인** 원칙을 따른다고 볼 수 있습니다. 이를 통해 자가 업데이트가 지나치게 보수적이지 않으면서도, 시스템 안정성을 해치지 않도록 균형을 맞춥니다.

시스템 흐름 및 테스트 전략

시스템 동작 흐름

자가 업데이트 엔진(`proposer.py`)과 파일 에이전트(`file_agent.py`)가 어떻게 상호작용하는지 **텍스트 다이어그램**으로 나타내면 다음과 같습니다:

1. **업데이트 감지 시작:** 일정한 주기 또는 사용자의 명시적 명령(예: `invoke auto.update`)에 따라 `proposer.py`가 실행됩니다. 실행 시, 엔진은 최신 LLM 지식, 패키지 인덱스(PyPI), 코드베이스 등을 **스캔**하여 업데이트 필요 항목을 수집합니다 (예: 사용 중인 패키지 버전 비교, 코드베이스의 DeprecationWarning 주석이나 호출 감지 등).

2. **개선 항목 분석:** 감지된 항목 각각에 대해, `proposer.py`는 어떤 **리팩토링 규칙**으로 처리해야 할지 결정합니다. 여기서 SELF_UPDATE_POLICY.md를 로드하여 각 항목에 해당하는 Rule의 **auto_approve 여부**와 **risk_level** 등을 조회합니다.

3. 예를 들어 `requests` 라이브러리 새 버전 발견 → `update_dependency` 규칙, deprecated 함수 `old_func()` 발견 → `replace_deprecated` 규칙, 코드 린트 오류 다수 발견 → `lint_fix` 규칙 식으로 매핑합니다.

4. 정책에 **Auto Approve = True**이고 현재 맥락에서 위험이 낮다고 판단되면, 바로 자동 적용 경로를 밟고, Auto Approve가 False이거나 위험 요소가 있다면 제안만 하고 넘어갑니다 (후속 manual 단계로).

5. **리팩토링 명령 생성:** 각 개선 사항에 대해 `proposer.py`는 대응되는 `invoke refactor` 명령을 구성합니다. 앞서 정의한 포맷대로 `--file`, `--rule`, 필요한 경우 추가인자를 조합합니다. 그리고 우선 **Dry-Run** 모드로 해당 명령을 실행하거나, 파일 에이전트의 함수를 직접 호출하여 **변경 계획(diff)**을 얻어냅니다.

6. Dry-Run 결과 diff는 로그에 저장하거나 사용자에게 요약 보고될 수 있습니다. 여러 개의 제안이 있을 경우, 이를 모아서 변경 내역 리스트를 작성합니다.

7. Auto Approve 대상인 경우 이 단계는 잠깐의 시뮬레이션에 불과하고 곧바로 다음 단계로 넘어가며, Auto Approve가 아닌 경우 여기서 diff만 제시하고 해당 제안은 **승인 대기** 상태로 남깁니다.

8. **파일 에이전트를 통한 수정 적용:** Auto Approve가 **True**인 변경사항에 대해서, `proposer.py`는 즉시 해당 명령을 **실행 모드**로 재호출합니다. 즉, `--yes` 플래그를 추가하여 `invoke refactor ... --yes` 형태로 파일 에이전트를 구동합니다 [5]. 파일 에이전트(`file_agent.py`)는 전달받은 인자를 파싱하여:

9. 지정된 규칙 모듈을 `get_rule`로 가져오고 [8], 대상 파일의 AST를 변환하여 새로운 코드로 적용합니다.
10. 이때 `--yes`가 포함돼 있으므로 실제 파일 쓰기가 수행되며, 적용 성공 시 표준 출력에 "**Successfully refactored**..." 메시지와 함께 diff 결과를 요약해서 보여줄 것입니다. (`--dry-run`인 경우 diff만 출력하고 파일은 그대로 둠).

11. 만약 주어진 규칙명이 잘못되었거나 AST 파싱/적용 중 예외가 발생하면, 파일 에이전트는 오류를 STDERR에 보고하고 해당 작업을 `returncode != 0`로 종료합니다. (예: 규칙 없음 -> "Unknown rule" 에러, 문법 오류 -> "Cannot parse file" 에러 등은 내부에서 처리되어 안전하게 종료됨을 테스트로 검증하고 있습니다 [9].)

12. **테스트 및 검증:** 파일 에이전트가 변경을 적용한 후, `proposer.py`는 SELF_UPDATE_POLICY의 지침에 따라 **후속 검증 절차**를 수행합니다.

13. 만약 해당 규칙에 `Test Required: True`로 지정되어 있다면, 즉시 프로젝트의 테스트 스위트를 실행합니다 (`invoke test` 등을 내부적으로 호출). **모든 테스트가 통과**하면 해당 변경을 확정짓고 다음 단계로 진행하지만, **테스트 실패** 시에는 곧바로 **롤백** 절차에 들어갑니다.
14. 롤백은 변경 이전 상태로 코드를 복구하는 것으로, 구체적으로는 `git`을 사용하는 환경이라면 `git stash` / `git reset` 혹은 커밋 단위의 revert를 시도할 수 있고, 그렇지 않다면 `file_agent.py`가 적용 전에 백업해둔 원본 파일을 덮어쓰는 방법 등을 고려합니다. (P1-2 단계에서 `--dry-run` 기능이 구현되어 있으므로, 실제 적용 전 diff를 확보한 상태이며, 필요한 경우 diff를 이용해 수동 revert도 가능하지만 가장 확실한 방법은 VCS를 통한 되돌리기입니다.)

15. 또한 Auto Approve였던 케이스라도 테스트를 통과하지 못하면 **자동 승인을 철회**하고, 해당 규칙의 `auto_approve` 설정을 재고하거나 추가적인 검토가 필요함을 로그로 남깁니다. 이런 상황은 **수동 검토 대상으로 강등**되는 셈입니다.

16. **결과 처리 및 보고:** 모든 후보 개선안에 대한 처리가 끝나면, `proposer.py`는 그 결과를 요약하여 출력하거나 내부 로그/리포트에 기록합니다. 자동으로 적용된 항목들은 "**Applied**"로 표시하고, 수동 검토로 남긴 항목들은 "**Pending Review**" 등으로 표시합니다. 예를 들어, `SELF_UPDATE_POLICY.md`에 Auto Approve가 False였거나 테스트 실패로 인해 적용 보류된 변화들은 사용자에게 “다음 변경 사항은 수동 승인이 필요합니다:”와 함께 diff나 간략 설명을 제공합니다. 필요시 `docs/HUB.md`의 Active Tasks에 “Self-Update: xxx 업데이트 수동 검토 필요”와 같은 항목을 등록해 개발자가 추적할 수 있도록 하는 방안도 고려합니다.

테스트 전략

통합된 Self-Update 시스템의 신뢰성을 확보하기 위해, **각 구성 요소별 및 통합 수준**에서 다음과 같은 테스트 전략을 수립합니다:

- **파일 에이전트 단위 테스트**: P1-2 파일 에이전트(`file_agent.py`)는 이미 기본적인 단위 테스트가 마련되어 있습니다. 예를 들어 `tests/test_p1_file_agent.py`에서 `--dry-run`시 파일 미변경 검증, `--yes`시 실제 적용 검증, 동일 규칙 반복 적용 시 멱등성 확인, 잘못된 규칙명 에러 처리, 경계 경로 차단, 문법 오류 예외 처리 등의 테스트가 구현되어 있습니다 [9]. 이러한 테스트 케이스들은 **모든 규칙 공통의 안전장치**를

점검하므로, 새로운 규칙을 추가할 때도 이들 테스트를 참고해 동일한 행동이 유지되는지 확인해야 합니다. 또한 각 개별 규칙 모듈(`rules/<rule_name>.py`)에 대한 별도 테스트도 작성하여, 특정 입력 코드에 대해 기대한 출력 코드가 나오는지 검사합니다.

- **Proposer 모듈 단위 테스트**: `proposer.py` 자체에 대한 테스트도 설계합니다. 여기서는 실제 외부 변화를 수반하지 않도록 **시뮬레이션된 환경**을 사용합니다. 예를 들어 가상의 `requirements.txt` 문자열과 최신 버전 정보를 넣어 주었을 때, `proposer.py`가 `update_dependency` 제안을 생성하는지 확인합니다. 또한 Deprecated 함수 호출이 있는 가짜 코드 조각을 입력으로 주고, 올바르게 `replace_deprecated` 규칙을 매핑하는지 테스트합니다. 이때 `SELF_UPDATE_POLICY.md` 파싱 로직도 함께 테스트하여, 예를 들어 특정 규칙의 `auto_approve=False` 설정을 바꿔가며 proposer의 동작(자동 적용 시도 여부)이 달라지는지를 검증합니다. 이러한 테스트들은 외부에 실제로 `invoke refactor`를 실행하지 않도록, `file_agent.py`의 인터페이스를 **mock**하거나 `--dry-run` 모드로 실행하도록 하여 결과만 받아 검사합니다.

- **통합 테스트 (End-to-End)**: 실제로 Self-Update 엔진이 파일 에이전트를 호출해서 코드를 고치는 일련의 과정을 통합 테스트합니다. 예를 들어 **시나리오**: 오래된 패키지를 포함한 임시 프로젝트 디렉터리를 만들고, 그 안에서 `invoke auto.update` (가정된 명령) 또는 `proposer.py`를 직접 실행합니다. 실행 후 `requirements.txt` 파일의 해당 패키지 버전이 올라갔는지, `git diff`로 변경 내역이 기대대로인지 확인합니다. 그리고 자동으로 `pytest`를 돌렸다면 그 결과도 캡쳐하여, 실패 시엔 롤백되었는지 또는 성공 시 커밋이 남았는지 등을 검증합니다.

- 한 예로, `requests==2.27.0`을 가진 requirements.txt와, 간단한 pytest가 통과하는 dummy 코드를 준비한 뒤 Self-Update를 돌려 **requests가 2.28.x로 올라가고 테스트가 통과**하여 최종 적용되는지를 본다.

- 반대로, 테스트가 실패하는 상황도 만들어봅니다. 의도적으로 Deprecated API를 바꿨을 때 한 곳의 수정 누락으로 테스트가 깨지게 한 뒤, Self-Update 엔진이 그 변경을 자동 취소하고 사용자에게 알리는지를 확인합니다. 이 과정에서 `proposer.py`가 **예외 처리**를 제대로 하는지 (예: file_agent 단계에서 예외가 던져져도 시스템이 중단되지 않고 다음 항목으로 넘어가는지)도 확인하게 됩니다.

- **정책 준수 테스트**: SELF_UPDATE_POLICY.md와 실제 동작의 일관성을 검증하기 위한 테스트도 필요합니다. 이는 일종의 설정-테스트로, 정책 파일에 명시된 모든 규칙에 대해 `proposer.py`가 알고 있는지, 그리고 policy의 값에 따라 conditional 분기가 잘 실행되는지 확인합니다. 예를 들어 정책에 `auto_approve: False`로 바꿔 넣고 돌렸을 때, 원래 자동 적용하던 시나리오에서 diff만 출력하고 멈추는지 테스트합니다. 또한 `risk_level`에 따른 로그 메시지나 사용자 경고가 제대로 표시되는지도 확인합니다 (예: 높은 위험도로 분류된 변경은 “manual review recommended” 식의 메세지 출력).

- **Fallback 및 수동 개입 절차 테스트**: 마지막으로, 자동화 프로세스에서 **downgrade to manual review** 경로를 철저히 시험합니다. 예를 들어, Self-Update 엔진이 처리하다가 예기치 못한 오류를 만나는 경우(네트워크 오류로 버전 확인 실패 등) 또는 테스트 실패/정책 불허로 자동 적용을 못하는 경우, 시스템이 이를 제대로 감지하고 **사용자에게 알림**하거나 **태스크를 보류**하는지를 본다. 구체적으로:

- 오류 상황 시 `proposer.py`가 Exception을 삼키지 않고 적절한 로그를 남긴 뒤 계속 실행되는지.
- 적용 보류된 변경이 HUB.md나 콘솔 출력에 “ 수동 검토 필요: ~~” 등의 형태로 표시되는지.
- (향후 개선) 보류된 제안에 대해 사용자가 원할 경우 수동으로 적용할 수 있도록, 예를 들어 `invoke refactor --file X --rule Y` 명령 가이드를 제공하는지도 확인합니다.

이러한 테스트 전략을 통해, 통합된 자기 업데이트 시스템이 작은 변경은 자동으로 잘 처리하면서도, 위험한 변경은 확실히 걸러내어 사람의 검토를 받도록 설계된 대로 동작하는지 검증합니다. 전체 기능이 자리잡으면, 정기적인 CI 파이프라인에서 `proposer.py`를 실행해봄으로써 실제 프로젝트에 미치는 영향과 테스트 통과 여부를 모니터링하고, 문제가 발생하면 즉시 피드백을 받아 정책을 조정하는 **Continuous Improvement** 사이클까지 이어갈 수 있을 것입니다.

1 3 6 8 9 P1-2_File_Agent_Framework_Upgrade.md

https://github.com/etloveaui/gemini-workspace/blob/517b10342b62d93c02d7e493b569b48a493ed2e6/docs/proposals/P1-2_File_Agent_Framework_Upgrade.md

2 PROJECT_ROADMAP.md

https://github.com/etloveaui/gemini-workspace/blob/517b10342b62d93c02d7e493b569b48a493ed2e6/docs/PROJECT_ROADMAP.md

4 5 7 HELP.md

https://github.com/etloveaui/gemini-workspace/blob/517b10342b62d93c02d7e493b569b48a493ed2e6/docs/HELP.md