ChatGPT

1. 아키텍처 검토

현재 제안된 플러그인 아키텍처는 `rules/__init__.py`에서 규칙 모듈들을 임포트하고 `RULES` 딕셔너리에 등록한 뒤, `get_rule(name)` 함수로 해당 규칙을 찾아오는 방식입니다. 이는 초기 구현의 **모놀리식 설계**(한 파일에 모든 규칙 로직 포함)를 개선하려는 시도로 보입니다. 실제로 현행 `file_agent.py`에서는 규칙 선택을 하드코딩(`--rule` 인자에 따라 if/else 분기)하고 있어 확장성이 낮습니다 [1]. 이러한 **RULES 딕셔너리 방식**은 구현이 간단하고 내부 규칙을 관리하기엔 충분하지만, 새로운 규칙을 추가할 때마다 수동으로 등록해야 하고 잘못된 키 입력시 런타임 에러가 날 수 있다는 단점이 있습니다. 또한 규칙 이름 충돌이나 로드 순서 관리에도 취약합니다.

Setuptools Entry Points 활용: 보다 유연한 플러그인 시스템으로는 setuptools의 엔트리 포인트(entry_points) 메커니즘이 있습니다. 이는 각 플러그인(예: 별도 패키지로 배포되는 규칙 모듈)이 자신을 특정 그룹에 등록해두고, 메인 프로그램이 `importlib.metadata.entry_points()`로 해당 그룹의 엔트리 포인트들을 조회하여 플러그인을 로딩하는 방식입니다 [2] [3]. 이 방법을 쓰면 외부 패키지나 플러그인을 **프로젝트에 추가 설치**만 하면 자동으로 발견되어 `RULES`에 등록되므로, 메인 코드 수정 없이도 규칙을 확장할 수 있습니다. 다만 이 접근은 패키징/배포가 전제되며, 현재 프로젝트가 내부 스크립트 중심이고 Windows 환경 특화라면 다소 **과잉 설계**가 될 수 있습니다. 또한 엔트리 포인트를 통한 플러그인은 실행 시 제3자의 코드를 로드하는 것이므로, **신뢰할 수 없는 코드 실행 위험**에 대비해 서명 검증이나 샌드박싱 등의 대책도 필요합니다 (물론 사내 플러그인만 쓴다면 리스크는 낮습니다).

클래스 기반 인터페이스 등록: 또 다른 대안은 **추상 클래스 기반**으로 규칙을 정의하고 자동 등록하는 것입니다. 예를 들어 `RuleBase` 추상 클래스를 만들어 모든 규칙이 이를 상속하도록 하고, `RuleBase`의 서브클래스가 정의될 때 자기 자신을 전역 레지스트리에 등록하도록 구현할 수 있습니다. 아래는 그런 구조의 예시입니다:

```
class RuleBase:
    registry = {}
    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        # 클래스 이름 또는 지정된 이름으로 자동 등록
        name = getattr(cls, "name", cls.__name__)
        RuleBase.registry[name] = cls

    def apply(self, source: str) -> str:
        raise NotImplementedError

# 사용 예시: 새 규칙 클래스 정의 시 자동 등록
class AddDocstringsRule(RuleBase):
    name = "add_docstrings"
    def apply(self, source: str) -> str:
        # ... AST 변환 로직 ...
        return new_source
```

이렇게 하면 `rules/__init__.py`에서 수동으로 딕셔너리를 관리하지 않아도, 프로그램 실행 시 `RuleBase.registry`를 순회하여 규칙들을 동적으로 로드할 수 있습니다. 장점은 **확장 시 실수로 RULES 등록을 빼먹는 문제를 방지**하고, 각 규칙이 자체 메타데이터(예: `name`, 지원되는 파라미터 등)를 클래스 속성으로 가질 수 있다는 것입니다. 또한 객체 지향 구조로 함으로써 규칙 간 공통 기능(예: 변환 전후에 공통 적용할 검사 등)을 `RuleBase`에서 제공하기도 쉽습니다. 단, 클래스 설계가 익숙하지 않은 팀원이나 작은 규모 스크립트에는 다소 부담스럽게 느껴질 수 있고, 초기 진입 장벽이 생길 수 있습니다.

런타임 플러그인 검색 (importlib.metadata / pkgutil): 엔트리 포인트를 사용하지 않더라도, Python 표준 모듈로 **동적 모듈 임포트**를 구현할 수 있습니다. 예를 들어 `rules` 폴더를 순회하며 `.py` 파일들을 찾고 (`pkgutil.iter_modules` 등 활용 [4]), 각각 `importlib.import_module`로 import한 뒤 규칙 객체를 수집하는 방법입니다. 또는 Python 3.10+에서는 `importlib.metadata`로 설치된 패키지의 메타데이터를 열람해 엔트리 포인트 없이도 비슷한 검색을 할 수 있습니다. 이런 **임포트 기반 검색**은 프로젝트 내부의 플러그인 구조를 갖출 때 유용하며, 규칙 파일을 추가하기만 하면 자동 인식하게 만들 수 있습니다. 예를 들면, `rules/__init__.py`에서 다음처럼 구현할 수 있습니다:

```
import importlib, pkgutil
RULES = {}
for finder, name, ispkg in pkgutil.iter_modules(__path__, prefix=__name__ + "."):
    module = importlib.import_module(name)
    if hasattr(module, "RULE"):   # 각 모듈에 RULE 정의
        rule = getattr(module, "RULE")
        RULES[rule.name] = rule
```

이 접근법의 이점은 **프로젝트 경계 내에서 확장성을 높여주고**, 규칙 모듈이 많아져도 중앙의 딕셔너리를 일일이 수정하지 않아도 된다는 점입니다. 반면, 규칙 모듈 import 시에 발생할 수 있는 예외 처리를 잘 해주어야 하며(어느 하나가 import 실패하면 전체 기능에 영향), 규칙 이름 중복에 대한 대비도 필요합니다. 또한 임포트 순서에 의존적인 전역 초기화 부작용이 없도록, 규칙 모듈은 import 시 가벼워야 합니다.

정리: 동적 로딩 방식(RULES 딕셔너리 또는 임포트 검색)은 현재 구조를 모듈화하여 확장성을 높이는 방향으로 적절합니다. 엔트리 포인트 방식은 **플러그인의 외부 배포**까지 고려할 때 유용하나, 현 시점에서는 내부 구조 개선만으로도 충분할 것입니다. 클래스 기반 설계는 규칙 관리의 견고성을 높이고 메타데이터 활용을 용이하게 하지만, 팀의 합의와 이해도가 필요합니다. 결국 중요한 것은 **규칙 추가/확장의 편의성과 안전성의 균형**이며, 현 단계에서는 **간단한 RULES 레지스트리 + 안전한 동적 임포트** 정도로 시작하고, 필요 시 점진적으로 고도화(예: entry point 지원)하는 전략이 바람직해 보입니다.

2. 핵심 질문에 대한 답변

2.1 AST 변환에서 포맷 보존과 주석 유지 방안

기본 AST 모듈의 한계: Python 내장 `ast` 모듈은 코드의 구조를 추출하기엔 좋지만, **소스의 포매팅과 주석 정보를 잃어버린다는 치명적 단점**이 있습니다 [5]. 예를 들어 `ast.parse()`로 코드를 파싱하면 공백, 들여쓰기 스타일, 주석과 같은 **비구조적 정보**는 AST에 포함되지 않습니다. 따라서 `ast`를 써서 코드 변환을 한 뒤 `ast.unparse()`나 `compile()`로 코드를 재생성하면 원래 포맷이 초기화되고 주석이 모두 사라집니다. 현재 `file_agent.py`의 `add_docstrings` 구현도 표준 AST를 사용하기 때문에, 추후 여러 규칙을 적용할 때 **변경과 상관없는 코드 스타일 변화나 주석 손실**이 발생할 수 있습니다 (이는 사용자에게 혼란을 주고 VCS(diff) 추적에도 불리합니다).

이를 해결하려면 **Concrete Syntax Tree(CST)** 기반의 도구를 고려해야 합니다. **LibCST** (Instagram/Meta에서 개발)는 대표적인 Python CST 라이브러리로, **모든 공백, 주석, 괄호까지 포함한 구문 트리를 제공**합니다 [6] [7]. LibCST를 사용하면 원본 코드와 바이트 단위로 동일한 출력까지 가능하며, 노드를 변환할 때 잘못된 구문을 만들면 바로 예외를 일으키는 등 안전장치도 갖추고 있습니다. 예를 들어 LibCST에서는 함수 정의 노드를 수정하더라도 내부 주석이나 문자열 리터럴의 원형을 유지한 채 변환할 수 있고, `Module.code_for_node()` 같은 메서드로 수정된 CST를 코드로 쉽게 출력할 수 있습니다. 요약하면 **LibCST는 AST의 사용 편의성과 CST의 포맷 보존력을 절충한 현대적인 솔루션**입니다.

다른 대안으로 **RedBaron** 라이브러리가 과거에 많이 거론되었습니다. RedBaron은 내부적으로 Baron이라는 CST 파서를 사용하여 **"Full Syntax Tree"** (FST)를 제공하고, 파싱한 노드 객체를 수정하면 자동으로 소스 코드에 반영되는

인터페이스를 갖추고 있습니다 [8]. 장점은 Python 코드를 텍스트로 다루듯이 노드를 추가/삭제할 수 있는 높은 직관성이었지만, **유지보수 문제**가 있습니다. Python3 지원이 한때 미흡했고(현재는 Python3 지원 버전이 나오긴 했으나), 핵심 파서(`baron`)의 버그가 충분히 해결되지 않은 채 프로젝트가 사실상 중단되어 있다는 지적이 있었습니다 [9]. 실제로 RedBaron을 쓰다가 새로운 Python 문법에 대응이 안 되거나 예기치 않은 FST 불일치 문제가 발생하면 직접 라이브러리를 수정해야 할 수 있어 리스크가 큽니다. **따라서 최신 Python까지 폭넓게 대응하려면 RedBaron보다는 LibCST를 사용하는 편이 안정적**입니다.

또한 **asttokens** 라이브러리는 기존 AST의 한계를 보완하기 위한 경량 도구입니다. 이 라이브러리는 파싱된 AST 노드들에 원본 소스의 토큰 정보를 주석처럼 붙여줌으로써, **각 AST 노드에 대응하는 정확한 소스 텍스트를 추출하거나 대응 토큰 목록을 얻을 수 있게 해줍니다** [10] [11]. 예를 들어 asttokens로 마크하면 어떤 노드의 `.first_token`과 `.last_token` 속성을 통해 해당 부분의 주석이나 공백을 포함한 텍스트를 가져올 수 있습니다. 이를 이용하면 ast를 써서 변환하면서도, 변경되지 않은 부분은 원본 토큰을 그대로 살려 출력하는 식의 구현도 가능합니다. 다만 asttokens는 **AST 자체를 CST로 바꾸는 것은 아니므로**, 직접 주석/공백을 재배치하거나 변환 로직마다 토큰을 관리하는 부담이 있습니다. 결국 asttokens는 **고급 변환을 위해 AST를 확장하는 저수준 도구**이고, 완전한 포맷 보존을 쉽게 달성하려면 LibCST 같은 CST 계열을 쓰는 편이 낫습니다.

Best Practice: 자동 코드 리팩터링 도구에서 **포맷과 주석을 최대한 보존**하는 것이 사용자 신뢰와 채택을 높입니다. 이를 위해 현재 AST 기반 접근을 계속할 경우, 최소한 asttokens를 도입해 **주석과 공백을 추적**하거나, 변환 후 원본 코드와 AST 출력코드를 `difflib`으로 비교해 **의도치 않은 변형이 없도록 검증**하는 절차가 필요합니다. 하지만 중장기적으로는 **LibCST로의 전환을 고려**하는 게 좋습니다. LibCST를 쓰면 각 변환 규칙을 일관된 Visitor/Transformer 패턴으로 구현할 수 있고, 여러 규칙을 적용해도 코드 스타일이 유지되므로 "코드가 깨끗이 바뀌는데 겉모습은 손대지 않은 듯"한 결과를 얻을 수 있습니다. 정리하면: **작은 변화**에는 표준 AST + asttokens 조합으로 세밀히 다루고, **큰 구조적 변경**이나 포맷 민감 작업에는 LibCST 같은 전용 라이브러리를 사용하는 혼합 전략도 고려할 수 있습니다.

2.2 자연어 기반 리팩터링 명령 → AST 전환을 위한 아키텍처 고려사항

Gemini-CLI의 목표 중 하나는 "이 파일의 모든 함수에 주석을 추가해줘" 같은 **고수준 자연어 명령을 이해하여** 실제 코드를 수정하는 것입니다 [12]. 이를 구현하려면 지금부터 **규칙 메타데이터 설계**, **프롬프트 템플릿 전략**, **AST ↔ 자연어 매핑 체계**를 염두에 두고 아키텍처를 준비해야 합니다.

규칙 메타데이터(schema) 설계: 각 리팩터링 규칙에 대해 자연어 인터페이스를 지원하려면, 규칙 자체에 **설명정보**가 포함되어야 합니다. 예를 들어 `AddDocstringsRule`이라면 `"모든 함수/클래스에 기본 docstring을 추가합니다"` 정도의 한줄설명과, 적용 범위나 전제조건(예: "이미 docstring이 있는 함수는 변경하지 않음") 등의 메모를 가질 수 있습니다. 이러한 메타데이터는 규칙 클래스의 속성이나 독스트링으로 저장해 둘 수 있고, CLI에서 `invoke refactor --list` 명령으로 규칙들과 설명을 출력하거나, 도움말에 포함시킬 수 있습니다. 더 나아가, **자연어 구문**과 규칙을 연결하기 위해 각 규칙에 키워드나 예시 문장을 첨부할 수도 있습니다. 예컨대 `RenameVariableRule`에는 `"rename <OldName> to <NewName>"` 같은 패턴을 등록해 두고, 자연어 명령 파싱 시 이 패턴에 매칭되면 해당 규칙을 선택하게 할 수 있습니다. 이러한 매핑 레지스트리를 마련해 두면, 규칙이 늘어나도 사용자가 **어떤 표현을 쓰면 어떤 규칙이 실행될지** 예측 가능하고, 시스템도 모호성 없이 명령을 해석할 수 있습니다.

Prompt 활용 전략: 만약 LLM 등 AI를 통해 자연어를 해석할 계획이라면, **프롬프트 템플릿**을 잘 구조화해야 합니다. 규칙 기반 리팩터링에서 LLM을 쓰는 방식은 크게 두 가지입니다: (1) **명령 분류/파라미터 추출** 용도로 LLM을 활용하는 것과 (2) **코드 수정 자체를 LLM에 요청**하는 것입니다. Gemini-CLI의 지향점이 "안전한 AST 기반 자동 수정"이므로, (2)보다는 (1)에 해당하는 사용이 바람직합니다. 즉, LLM은 자연어 명령을 읽고 "어떤 규칙(rule)을 어떤 인자와 함께 실행해야 목표를 달성할까?"를 판단하는 역할만 맡고, 실제 코드 AST 변환은 검증된 규칙 로직이 수행하는 것입니다. 이럴 경우 프롬프트에는 **사용 가능한 규칙 목록과 설명**을 포함시켜, 모델이 답변으로 `<규칙 이름>: 인자들` 형태의 **구조화된 출력**을 내놓도록 유도하면 됩니다. 예를 들어 프롬프트에 `"사용가능한 기능: add_docstrings(모든 함수에 주석 추가), rename_variable(X를 Y로 변수명 변경), ... 사용자 요청: 'Foo 함수를 제거하고 호출부분을 로그로 남겨줘'"`라고 주고, 모델이 이에 대해 `"remove_function(name='Foo', replacement='log call')"`, 혹은 `"규칙 없음"`과 같이 답하게 할 수 있습니다. 이를 위해서는 **규칙 메타데이

터에 앞서 언급한 자연어 설명이 충실히 작성되어야 하고, 모델이 출력할 포맷도 미리 약속되어야 합니다 (예: JSON이나 `<rule>(param=...)` 형태 등).

Prompt 설계의 또 다른 측면은 **추론 과정의 투명성**입니다. 자연어 명령을 바로 LLM에게 던져 규칙 이름을 받는 것보다, 단계적으로 "사용자 명령 분석 → 해당하는 규칙 찾기 → 규칙의 파라미터 채우기" 과정을 프롬프트로 풀어쓰면 모델이 실수를 줄일 수 있습니다. 예컨대 체인드 프롬프트를 통해, 첫 번째 질문으로 “사용자 의도가 다음 중 어떤 작업과 부합하는가? (A: 함수추가, B:변경, C:삭제...)”를 묻고, 두 번째로 세부사항을 채우는 식입니다. 다소 복잡해보이지만, 이러한 **프롬프트 체계**를 미리 구상해두면 구현 단계에서 시행착오를 줄이고, 메타데이터 스키마도 이에 맞게 설계할 수 있습니다. 만약 프롬프트 없이 규칙 매핑을 하더라도, 정규식이나 심볼릭 AI를 이용해 자연어를 해석해야 하므로, 역시 규칙별 키워드 리스트나 문맥 정보가 필요합니다.

AST ↔ NL 매핑 레지스트리: 장기적으로는 자연어 명령과 AST 패치 간의 **양방향 매핑**이 가능하도록 구조를 짜두면 혁신적인 기능을 추가할 수 있습니다. 예를 들어, 규칙이 적용된 후 사용자에게 "X 규칙을 적용하여 5개의 함수를 수정했습니다. (모두 docstring 추가)"와 같이 **자연어 보고**를 할 수 있을 것입니다. 이를 위해 각 규칙은 자기 변화의 의미를 한 줄 설명할 수 있어야 하고(예: “Added docstrings to N functions”), 규칙이 변경한 AST 노드와 원본 소스 라인을 연결짓는 메커니즘이 있으면 좋습니다. AST 툴에서 각 노드에는 위치 정보가 있으므로 (예: ast.Node.lineno 등), 이걸 기반으로 어떤 코드가 어떻게 변형됐는지 추적해 **자연어 Diff**를 만드는 것도 고려해볼 수 있습니다. 예컨대 “라인 10: 함수 foo()에 docstring 추가” 같은 설명을 자동 생성하는 식입니다. 이러한 매핑 레지스트리는 단순히 사용자 편의를 넘어서, 나중에 AI가 자체 코드를 수정하고 스스로 그 결과를 평가하는 **자가 개선 루프**에도 활용될 수 있는 데이터가 됩니다. 그러므로 지금 단계에서 규칙 구현 시 **"이 규칙은 무엇을, 왜 하는가"**에 대한 메타 정보를 축적해두면 향후 큰 자산이 될 것입니다.

정리하면, 자연어 기반 리팩터링을 위해 **지금부터 해야 할 일**은: 1) 각 규칙에 풍부한 설명과 키워드를 부여하고, 2) 자연어 명령을 규칙으로 변환하는 체계를 설계하며 (규칙 선택 및 파라미터 추출을 어떻게 할지), 3) 규칙 적용 결과를 다시 자연어로 표현하거나 학습에 활용할 수 있도록 AST와 NL의 매핑 정보를 남기는 것입니다. 이러한 준비가 뒷받침되면, 추후 간단한 명령부터 복잡한 리팩터링 시나리오까지도 **AI를 통한 자동화**가 매끄럽게 이루어질 것으로 기대됩니다.

3. 코드 레벨의 개선 제안

현재 `file_agent.py` 및 향후 추가될 `rules` 모듈 구조를 좀 더 **견고하고 테스트하기 쉽게** 리팩터링하기 위한 몇 가지 제안을 드리겠습니다:

- **RuleBase 클래스 도입 및 규칙 모듈화:** 앞서 아키텍처 검토에서 언급했듯, 규칙들을 객체로 추상화하면 코드 구조가 훨씬 깔끔해집니다. `file_agent.py`의 `add_docstrings` 함수는 지금은 단순 함수이지만, 이를 `AddDocstringsRule` 클래스의 `apply()` 메서드로 바꾸고 `RuleBase`를 상속하도록 리팩터링하세요. 이렇게 하면 규칙 별로 독립된 모듈 (`rules/add_docstrings.py` 등)로 분리하기 쉽고, 각 클래스 자체를 단위 테스트할 수 있습니다. 예를 들어 `AddDocstringsRule.apply()`에 대해 다양한 함수 정의가 들어있는 문자열을 주입해 결과를 검증하는 테스트를 작성하면, 파일 I/O 없이도 로직 검증이 가능합니다. 또한 클래스 구조를 사용하면 규칙 간 **공통 지원 기능**을 구현하기 좋습니다. 예를 들어 모든 규칙은 `before_apply(source)`와 `after_apply(new_source)` 훅을 가져서, 적용 전후에 코드의 유효성을 검증하거나 로그를 남기는 기능을 일괄 추가할 수 있습니다. 이런 훅은 RuleBase 기본 구현으로 두고 개별 규칙이 필요시 오버라이드하면 됩니다. 결과적으로 클래스 기반 리팩터링은 **규칙 추가 시 boilerplate을 줄이고**, 잘못된 사용을 컴파일 타임에 발견하며(없는 속성 접근 등), 코드 가독성을 높여줄 것입니다.

- **명시적 I/O 분리:** `file_agent.py`의 현재 구현에서는 파일을 읽고 쓰는 로직과 AST 변환 로직이 한 함수 안에 섞여 있습니다. 이를 **입출력과 비즈니스 로직의 분리** 원칙에 따라 재구성해야 합니다. 권장 방식은, 실제 변환은 **순수 함수**로 만들고 파일 접근은 호출부에서 처리하는 것입니다. 예를 들어 `apply_rule_to_code(source_code: str, rule_name: str) -> str` 형태의 함수(또는 RuleBase.apply 호출)를 만들고, 이 함수는 입력 문자열을 받아 변환된 문자열을 리턴하기만 하도록 합니다. 그리고 파일 읽기/쓰기, diff 생성, 사용자 confirm 받기는 바깥 레벨에서 담당합니다. 이렇게 하면 변환 자체는

언제든지 **dry-run 모드**로 호출하여 결과 문자열만 비교해볼 수 있고, 파일 시스템에 영향이 없으므로 단위 테스트가 용이해집니다. 이미 `add_docstrings` 함수는 입력/출력 분리가 되어 있으니, 이를 발전시켜 **모든 규칙이 입력 문자열 -> 출력 문자열만을 다루도록 규약**을 만들면 좋습니다. 아울러 `invoke refactor` 커맨드에서 다룰 인자들도 확장에 대비해 유연하게 설계해야 합니다. 현재는 `--file` 하나지만, 추후 규칙에 따라 추가 파라미터(`--old-name`, `--new-name` 등)가 필요할 수 있으므로, Invoke 태스크 시그니처나 `file_agent.py`의 argparse 정의를 **규칙별 매개변수 플러그인** 형태로 바꾸는 것도 고려해야 합니다. (예: 규칙 클래스로부터 argparse subparser를 생성하거나, 규칙 실행 전에 파라미터 검증을 하는 구조)

- **Dry-run 출력 개선 (컬러화 등):** Dry-run 모드에서 출력되는 unified diff는 현재 흑백 텍스트로만 제공됩니다[13]. 사용자가 변경점을 한눈에 파악하려면 **색상 하이라이트와 서식 개선**이 필요합니다. 다행히 이미 프로젝트에 `rich` 라이브러리가 도입되어 있으므로[14], 이를 활용하면 터미널 컬러 출력은 쉽게 구현 가능합니다. 예를 들어 `rich.console.Console().print()`로 diff 텍스트를 출력하면서, 줄이 `+`로 시작하면 녹색으로, `-`로 시작하면 빨간색으로 스타일링할 수 있습니다. 또는 `rich.markup`을 사용해 `[-]`/`[+]` 마크업 태그로 색을 지정해도 됩니다. 더 나아가 `rich.diff.Diff` 객체를 활용하면 two-column 컬러 diff까지도 구현 가능하지만, 우선은 **라인 단위 색상 표시**만으로도 가독성은 크게 올라갈 것입니다. 이 때 Windows-first 환경을 고려하면, ANSI 컬러 코드 출력 시 호환성을 신경써야 하지만, `rich`는 내부적으로 Windows 터미널 처리도 지원하므로 큰 문제는 없을 것입니다. 추가로, Dry-run일 때 현재는 diff만 보여주는데, **요약 정보**를 함께 주면 좋습니다. 예컨대 "총 5줄 변경 예정 (+3 / -2)" 같은 통계를 diff 앞이나 뒤에 출력하면 사용자가 변경 규모를 파악하기 쉽습니다. 이러한 정보는 `difflib.unified_diff` 결과를 한 번 훑거나, AST 변환 결과와 원본을 비교하여 계산할 수 있습니다. 작은 UX 개선이지만 사용자 신뢰를 높이는 요소입니다.

- **사용자 프롬프트/확인 로직 개선:** 현재 `confirm_action`을 통해 Y/N 확인을 받고 있는데[15], 기본값이 "no"인 점은 안전을 위해 바람직합니다. 다만 테스트 자동화를 위해 **비대화형 모드**에서도 동작하게 할 필요가 있습니다. 예를 들어 `invoke refactor --dry-run`은 자동으로 통과되겠지만, `--yes` 같은 옵션을 추가로 받아서 주어진 경우에는 사용자 확인 없이 곧바로 적용하도록 하면 CI나 스크립트에서 활용하기 좋아집니다. 또는 환경변수로 `CONFIRM=0` 등을 읽어 자동화할 수도 있습니다. 한편, 프롬프트 자체도 `prompt_toolkit`을 활용하고 있으므로, 단순 Y/N 입력 대신 **화살표 키로 Yes/No 선택**하는 UX 개선이나, 타임아웃 후 자동 취소 등의 기능도 고려할 수 있습니다 (P2 단계의 UX 향상 목표와 맥락을 같이합니다[16]). 이러한 개선은 필수 기능은 아니지만, **사용자 경험을 세밀하게 다듬어 신뢰감을 주는 효과**가 있습니다.

- **코드 중복 및 구조 개선:** `file_agent.py`를 살펴보면 dry-run과 실제 적용에서 diff를 생성하는 부분이 중복되어 있습니다[13][17]. 이 코드를 함수로 뽑아내어 `display_diff(old:str, new:str, file:str)` 형태로 만들면 중복을 제거하고 의도를 드러낼 수 있습니다. 마찬가지로, `print("The following changes will be applied:")` 등의 메시지도 하드코딩되어 있는데, 다국어 지원이나 UI 개선을 고려하면 이 부분을 `runner.py`의 공용 함수나 `locales` 설정으로 옮겨두는 것이 일관성에 좋습니다. 전반적으로 **한 가지 작업은 한 곳에서만 구현**되도록 리팩터링하면, 향후 유지보수성과 테스트 용이성이 크게 향상됩니다. 예컨데, confirm_action 이후 파일을 쓰는 로직도 지금은 file_agent에 직접 있지만, 이것을 `runner.apply_changes(path, new_code)` 식으로 래핑하면 파일쓰기 직전의 공통 처리(백업 떠놓기 등)도 일괄 추가하기 쉽겠죠. 작은 프로젝트라 간소하게 간 것도 이해되지만, 이제 기능이 늘어나는 시점이므로 **지금 리팩터링해 두면 기술 부채가 쌓이는 것을 방지**할 수 있습니다.

4. 잠재적 위험 분석

마지막으로, 현재 구조와 계획된 기능을 토대로 **잠재적 위험 요소**들을 짚어보고 대응 방안을 제시합니다:

- **보안 측면 – 경로 탐색 및 프로젝트 경계 이탈:** `invoke refactor --file` 인자로 임의의 경로를 받는 현재 설계에서는, 악의적이거나 부주의한 입력으로 **프로젝트 밖 파일을 수정하는** 일이 생길 수 있습니다. 예를 들어 `--file "../settings.py"` 같은 인자를 주면 상위 디렉토리의 파일도 접근 가능합니다. 실제 코드

에서 `file_path = ROOT / args.file`로 경로를 만들지만 [18], Python의 `pathlib`는 인자로 절대경로가 들어오면 `ROOT`를 무시하고 그대로 사용해버리므로 이 경우도 통과됩니다. 이는 **프로젝트 경계 정책**을 어기는 위험으로, 자칫하면 전혀 다른 경로의 중요 파일을 수정하는 보안 사고로 이어질 수 있습니다. 대응 방안으로는, `file_agent.py`에서 파일 지정 시 **루트 경로 하위 제한**을 거는 것입니다. `file_path = (ROOT / args.file).resolve()` 한 뒤에 `if ROOT not in file_path.parents: ...` 형태로 체크하여, 워크스페이스 밖을 벗어난 경로이면 바로 에러를 내고 중단해야 합니다. 또한 `..` 시퀀스를 포함한 경로는 애초에 받지 않도록 argparse 단계에서 validation을 넣을 수도 있습니다. 두 번째 보안 요소는 **무분별한 파일 수정**입니다. 규칙에 따라 광범위한 변경이 발생할 수 있는데, 사용자가 의도를 정확히 파악하지 못한 상태에서 실행하면 대량의 코드 변경이나 손실이 발생할 수 있습니다. 이를 막기 위해 이미 dry-run + 확인 단계를 넣었지만, 추가로 고려할 것은 **백업/Undo 메커니즘**입니다. 예를 들어 변경 적용 전에 대상 파일의 백업본을 `.bak` 확장자로 저장해두거나, Git을 사용 중이라면 자동 커밋/스태시를 떠놓고 진행하면 최악의 경우에도 원상복구할 수 있습니다. 현재는 사용자가 수동으로 Git 관리를 해야 하지만, 도구가 한 발 더 나아가 **"안전망"**을 제공하면 신뢰도가 높아집니다. 마지막으로, 플러그인 규칙을 동적으로 로딩하는 경우 **임의 코드 실행 위험**이 있습니다. 만약 규칙 모듈이 악성 코드를 내포하거나, 규칙 이름을 가장한 잘못된 입력이 import 경로 조작에 악용되면 보안 문제가 될 수 있습니다. 이를 대비해 `rules` 폴더 외부의 모듈 import는 시도하지 않도록 철저히 통제하고, 필요하면 로딩 시 sandbox나 permission 제한을 거는 것도 검토하세요.

- **유지보수 측면 – 규칙 간 의존성과 테스트 부족:** 규칙이 하나일 땐 문제가 없지만, 여러 규칙을 운용하게 되면 **규칙 간 충돌이나 의존성**이 생길 수 있습니다. 예를 들어 "포매터(rule_A) 적용 후 변수이름 변경(rule_B)" 순서로 해야 하는데 거꾸로 하면 안 된다든지, `rule_A`가 이미 코드를 바꿔놓아서 `rule_B`의 매칭 로직이 실패한다든지 하는 시나리오입니다. 현 단계에선 한 번에 하나의 규칙만 적용하지만, 사용자가 연속된 명령을 내리거나 복합 명령("이 함수 이름 바꾸고 docstring도 추가")을 내릴 가능성을 고려해야 합니다. 이를 위해 **규칙 간 우선순위나 전제조건을 명시**하는 체계를 갖추는 것이 좋습니다. 예컨대 RuleBase에 `pre_conditions`나 `conflicts` 필드를 두어, 특정 규칙이 선행되어야만 동작하는 경우나 함께 적용하면 안 되는 조합을 선언할 수 있습니다. 또한, 각 규칙은 **가능한 한 독립적**으로 설계해야 합니다. 한 규칙이 다른 규칙의 내부 함수나 전역 변수에 의존하면 추적이 힘들어지고 버그를 유발합니다. 이를 피하려면 규칙 간 데이터를 공유하지 않고, 필요한 경우 명시적으로 상위 레벨에서 결과를 전달하도록 해야 합니다.

유지보수에서 더 큰 위험은 **테스트 미비**입니다. 리팩터링 도구 특성상, 작은 실수로도 코드베이스에 광범위한 영향을 줄 수 있으므로 철저한 테스트가 필요합니다. 현재 일부 기본 테스트는 있는 것으로 보이나 [19], 규칙 추가 시마다 해당 규칙의 단위 테스트를 작성하고 다양한 시나리오를 점검해야 합니다. 예를 들어 `add_docstrings`의 경우, 이미 docstring이 있는 함수, 여러 타입의 함수(def, async def, lambda 등), 클래스 내부의 함수 등 다양한 입력에 대해 **예상대로 동작하는지** 테스트해야 합니다. 또 한 가지 중요한 테스트는 **idempotency (멱등성)**입니다. 한 규칙을 두 번 적용해도 결과가 처음 한 번 적용한 것과 같아야 하는데, 이를 확인하지 않으면 반복 실행 시 코드가 조금씩 변형되는 버그가 생길 수 있습니다. (예: `add_docstrings`를 두 번 실행했더니 docstring이 이중으로 붙는다든지). 이러한 부분까지 자동화된 테스트로 검증해두면 유지보수 시 마음 놓을 수 있습니다. 마지막으로, **CI 파이프라인**에 이 테스트들을 포함시켜 Windows/Unix 환경 모두에서 통과되도록 하는 것이 중요합니다. 현재 Windows 우선 개발이라도, CI에서 다양한 환경을 확인해두면 나중에 호환성 문제를 조기에 발견할 수 있습니다.

- **UX 측면 – 규칙 적용 조건 불명확 및 오류 대응:** 사용자 경험 상의 위험으로는, **규칙 동작의 불명확함**이 있습니다. 사용자가 어떤 리팩터링을 요청했을 때, 해당 규칙이 어떤 조건에서 작동하고 어디까지 영향 미치는지 모르면 혼란을 겪을 수 있습니다. 예를 들어 `add_docstrings` 규칙은 "모든 함수에 주석 추가"라고 하지만, 실제론 이미 주석이 있는 함수는 건드리지 않습니다. 만약 사용자가 이를 모르고 "왜 일부 함수는 안 바뀌었지?"라고 생각한다면 UX 실패입니다. 대응책은 **명확한 문서화와 피드백 메시지**입니다. CLI 도움말이나 문서(`README` 혹은 `HELP.md`)에 각 규칙의 동작을 상세히 설명하고, 실행 시에도 중요한 전제사항을 출력해주는 게 좋습니다. 예컨대, `No changes to apply.`라고만 하지 말고 "No changes - all functions already have docstrings."처럼 이유를 알려주면 사용자 이해도가 높아집니다. 또한, **사용 가능한 규칙 목록과 사용법 노출**도 UX에 중요합니다. 현재는 잘못된 규칙명을 넣으면 "Unknown rule" 에러만 나오는데 [20], 차라리 사용 가능한 규칙들을 나열해 주는 편이 친절합니다. (예: "Unknown rule 'foo'. Available rules:

add_docstrings, rename_variable, ..."). 추후 규칙이 많아지면 `invoke refactor --help`나 별도 `invoke refactor --list`에 이 정보를 제공해 검색할 수 있게 하세요.

또 다른 UX 위험은 **오류 상황에서의 처리**입니다. 만약 AST 파싱 오류나 규칙 로직 에러로 리팩터링에 실패하면, 지금은 아마도 traceback을 출력하고 죽을 것입니다. 이는 사용자를 당혹시키므로, **예외 처리**를 세분화해야 합니다. 예를 들어 파싱 오류(구문 오류 있는 파일)에 대해서는 "파일을 파싱하지 못했습니다. 문법 오류를 확인해주세요.", 규칙 로직의 KeyError 같은 예외는 "리팩터링 규칙 실행 중 오류 발생: ... (개발팀에 문의)" 식으로 구분해 알려주면 됩니다. 또한 P2 목표에 언급된 대로, 오류 시 과거 로그나 해결책을 제시하는 기능까지 넣으면 금상첨화입니다 [16]. 사용자 입장에서는 **실패도 친절하게** 안내해주는 도구에 신뢰를 느끼게 됩니다.

끝으로, **자연어 명령 UX**에 대한 잠재 이슈도 생각해야 합니다. 향후에는 사용자가 진짜 대화형으로 "이 코드좀 정리해줘"라고 입력하면, 시스템이 적절한 규칙을 골라 수행하는 형태가 될 텐데, 이때 **모호한 명령 처리**가 난관입니다. 사용자 의도가 불분명하면 잘못된 규칙을 적용할 위험이 있습니다. 이를 줄이려면, 가능하면 사용 단계에서 추가 확인을 하는 것도 방법입니다. 예를 들어 "함수 정리해줘" 라는 명령에 여러 후보 규칙이 떠오른다면, 한 단계 물어보는 겁니다: "함수명을 변경하려는 것인가요, 아니면 함수 내용을 리포매팅하려는 건가요?". 이런 **디스암버iguation 대화**를 지원하면 오작동을 줄일 수 있지만, 구현 난이도가 높으므로 장기 과제로 고려하세요. 단기적으로는 아키텍처 차원에서 **모델이 모호한 응답을 내놓지 않도록** 프롬프트를 촘촘하게 짜거나, 규칙 매칭 알고리즘을 엄격히 하는 식으로 대비해야 합니다.

요약하면, 보안/유지보수/UX 각 측면에서 **"방어적인 설계"**가 필요합니다. 입력 검증을 철저히 하고, 규칙 간 관계와 테스트 커버리지를 강화하며, 사용자에게는 가능한 한 많은 정보를 제공하여 신뢰를 주는 것이 중요합니다. Gemini-CLI는 개발 보조 도구인 만큼, **한 번의 실수가 큰 피해**로 이어질 수 있습니다. 그러므로 지금 제시된 개선책들을 순차적으로 도입해간다면, 더욱 안정적이고 사랑받는 리팩터링 도구로 발전할 것으로 기대합니다.

---

[1] [13] [15] [17] [18] [20] file_agent.py
https://github.com/etloveaui/gemini-workspace/blob/8783a95588b00d690a2e630689b920f1a9f150d0/scripts/file_agent.py

[2] [3] [4] Creating and discovering plugins - Python Packaging User Guide
https://packaging.python.org/en/latest/guides/creating-and-discovering-plugins/

[5] [8] [9] Python AST with preserved comments - Stack Overflow
https://stackoverflow.com/questions/7456933/python-ast-with-preserved-comments

[6] [7] Why LibCST? — LibCST documentation
https://libcst.readthedocs.io/en/latest/why_libcst.html

[10] [11] API Reference — asttokens documentation
https://asttokens.readthedocs.io/en/latest/api-index.html

[12] [16] [P1_P2]Plan_Gemini.md
https://github.com/etloveaui/gemini-workspace/blob/8783a95588b00d690a2e630689b920f1a9f150d0/scratchpad/Gemini-Self-Upgrade/Plan/[P1_P2]Plan_Gemini.md

[14] 20250805_Daily_Log.md
https://github.com/etloveaui/gemini-workspace/blob/8783a95588b00d690a2e630689b920f1a9f150d0/scratchpad/20250805_Daily_Log.md

[19] [P0]Debug_12.md
https://github.com/etloveaui/gemini-workspace/blob/8783a95588b00d690a2e630689b920f1a9f150d0/scratchpad/Gemini-Self-Upgrade/[P0]Debug_12.md