# ChatGPT

## 핵심 파일 식별 (파일 수정 및 재시도 관련)

분석 대상 레포지토리에서 **파일 수정 작업**과 **도구 실행/오류 처리/재시도** 로직을 담당하는 주요 컴포넌트는 다음과 같습니다:

- `tasks.py` (프로젝트 루트): CLI 명령을 정의하는 파일로, `invoke` 명령을 통해 에이전트 작업이나 파일 수정을 실행합니다. 예를 들어 `text.replace` 작업을 정의하여 내부적으로 `scripts/textops.py`를 호출하고 있습니다 [1]. 또한 `edits` 네임스페이스를 통해 파일 수정 제안/적용(workflow) 작업들을 정의합니다 [2] [3].
- `scripts/textops.py`: 텍스트 치환 등의 **파일 내용 수정 도구**를 제공하는 스크립트입니다. 특히 `replace_with_lineending_tolerance()` 함수는 파일의 줄바꿈 형식(CRLF/LF)을 **정규화**하여 문자열 치환의 신뢰성을 높이는 로직을 구현하고 있습니다 [4]. 이 함수는 파일에서 `old` 문자열을 찾아 `new`로 교체하며, 치환 발생 횟수를 반환합니다. 치환 대상이 없으면 0을 반환하고 종료하여 (예외를 발생시키지 않고) **"0건 치환"**으로 간주합니다 [4]. 치환이 발생한 경우에만 파일을 덮어쓰도록 구현되어 있어, 불필요한 파일 쓰기를 피합니다 [5].
- `scripts/edits_manager.py`: 멀티에이전트 시나리오에서 **안전한 파일 수정 워크플로**를 담당합니다. 에이전트의 수정 제안을 파일별 "프로포절(proposal)"로 캡쳐(`capture`), 수정내용 작성(`propose`), 변경점 비교(`diff`), 실제 적용(`apply`) 등의 서브커맨드를 가지며, 각각 CLI로 호출 가능합니다 [2] [6]. 특히 `cmd_apply` 함수는 제안된 변경을 실제 파일에 적용하며, 적용 전에 **안전 검사**를 수행합니다. 이 안전 검사가 바로 중복 수정 방지 및 재시도 제어 로직의 핵심입니다 [7] [8].
- `scripts/tools/edits_safety.py`: `edits_manager`의 적용(`apply`) 단계에서 호출되는 **중복 방지 및 백오프 전략**을 구현한 모듈입니다. 최근의 파일 수정 시도를 기록하고(`record_result`), 동일한 패치가 짧은 기간 내 반복되는지를 확인하는 `should_apply` 함수를 제공합니다 [9] [10]. 이 모듈은 `.agents/edits_state.json` 상태 파일을 통해 기록을 영구 저장하며, 향후 동일한 요청이 들어왔을 때 적용 여부를 판단합니다.

위 파일들이 연계되어 GEMINI CLI 기반의 파일 수정 시나리오에서 문제의 원인이 된 **"실패한 동일 요청의 반복"** 이슈를 완화하도록 구성되어 있습니다. 다음 섹션에서는 이들 파일 내 **주요 함수와 제어 흐름**을 분석합니다.

## 수정 도구 호출부 핵심 함수 및 제어 흐름

**1. CLI 태스크(trigger) → Textops 도구 실행:** 사용자가 GEMINI CLI에서 파일 치환 명령을 내릴 때 (`invoke text.replace ...` 등), `tasks.py`의 해당 함수가 호출됩니다. 예를 들어 `text_replace` 함수는 `scripts/textops.py`를 **별도 프로세스로 실행**하여 `replace` 작업을 수행합니다 [1]. 이 함수는 인자로 받은 파일 경로(`--file`), 교체 대상 문자열(`--old`), 교체할 문자열(`--new`) 등을 `textops.py`에 전달하고, `check=False`로 실행하여 서브프로세스 오류를 잡지 않고 계속 진행합니다 [1].

**Textops 내부 흐름:** `scripts/textops.py`의 `main()` / `cmd_replace`가 실행되면, 실제 치환 로직인 `replace_with_lineending_tolerance()`가 호출됩니다. 이 함수는: (a) **파일 바이트를 읽어 줄바꿈 스타일 감지**, (b) 입력 문자열 `old` / `new` 및 파일 내용을 LF 기준으로 **정규화** [4], (c) `old` 패턴의 등장 횟수를 계산합니다. 기대하는 치환 횟수(`--expect` 인자)가 있으면 검증하고, 그렇지 않은 경우라도 **등장 횟수가 0이면 즉시 0을 반환**하여 종료합니다 [4]. 등장 횟수가 0이라는 것은 파일에 `old` 문자열이 없어서 수정할 필요가 없음을 뜻하므로, 이 때는 파일에 아무 변화도 주지 않고 종료합니다. 만약 1회 이상 등장하면 모든 발생을 `new`로 치환한 새로운 내용(`updated_lf`)을 만든 뒤, **원본 파일의 줄바꿈 스타일**로 변환하여 덮어쓰기 합니다 [5]. 치환 결과(치환된 건수 `n`)는 `cmd_replace`를 통해 출력되며, 예외 상황에서는 에러 메시지를 출력하고 프로세스를 종료하도록 되어 있습니다 [11].

**오류 처리:** `textops.py`는 치환 건수가 0인 경우도 정상 동작으로 간주하기 때문에 예외를 던지지는 않습니다 (이때 `n=0`으로 출력). 그러나 `--expect` 옵션으로 예상 치환 수를 지정했고 실제 치환 수가 맞지 않으면 `ValueError`를 일으켜 `SystemExit(1)`로 종료합니다 [12] [13]. 또한 파일 쓰기 중 문제가 생기면 예외가 발생하여 `SystemExit(1)`로 이어집니다. 이러한 오류 발생 시 `tasks.py`의 `run_command` 래퍼는 `check=False` 설정으로 인해 예외를 전파하지 않고, 대신 **프로세스 반환코드와 stdout/stderr 정보를 수집**합니다. (참고로 `scripts/runner.py`의 `run_command` 구현은 `check=True`일 때 예외를 잡아 DB에 기록하고 콘솔에 패널로 표시하는 로직이 있지만 [14], 여기서는 `False`이므로 오류를 단순히 결과로 담고 넘어갑니다.)

- **재시도 로직:** 현재 코드 상에서는 `textops` 레벨에서 실패 시 **자동 재시도**를 수행하지 않습니다. 다만 GEMINI 에이전트(LLM)가 동일 명령을 다시 보내는 방식으로 재시도가 발생하는데, 이때 이전 시도와 동일한 요청이라면 이를 인지하여 막는 쪽이 필요합니다. 이러한 맥락에서, 레포지토리 측에서는 개별 도구가 재시도를 직접 관리하기보다는 **상위 레벨에서 중복 시도를 감지/차단**하도록 설계했습니다.

**2. EditsManager를 통한 안전한 수정 적용 흐름:** 에이전트가 곧바로 `text.replace`를 호출하는 대신, **사전 검증된 패치 적용 워크플로우**인 `edits_manager`를 사용할 수도 있습니다. 이 경우 제미나이 에이전트는 먼저 `edits.capture`로 대상 파일의 현재 내용을 `.edits/proposals/<file>`로 캡쳐하고, 수정할 내용을 반영한 새로운 파일 내용을 `edits.propose`로 저장합니다 [15] [16]. 그런 다음 `edits.diff`로 변경사항을 확인하고 `edits.apply`를 실행하면, `cmd_apply` 함수가 실제 파일 수정 적용을 시도합니다.

`cmd_apply`의 **제어 흐름**은 다음과 같습니다 [17] [7]:
① 적용 대상 파일별로, 캡쳐된 원본과 제안된 수정본 간의 **diff(차이)**를 계산합니다. diff 내용이 없다면 (즉, 실제 변경점이 없다면) 해당 파일은 건너뜁니다.
② diff가 존재할 경우, 이를 **SHA-256 해시**로 계산하여 패치 식별자로 삼습니다. 그런 다음 `edits_safety.should_apply()` 함수를 호출하여 **이 패치를 지금 적용해도 되는지 확인**합니다 [7]. 이 함수 내부에서 곧 설명하겠지만, 최근에 동일한 패치가 성공적으로 적용되었거나 반복 실패한 이력이 있으면 적용을 **보류**합니다.
③ `should_apply`가 `True`를 반환하면 (즉, 적용해도 안전하다고 판단되면) diff 내용을 콘솔에 출력하여 사용자에게 확인을 받습니다. (자동 확인 모드가 아니면 `y/N` 입력 대기)
④ 사용자가 승인을 하면, **실제 파일을 덮어쓰기**하여 수정 내용을 반영합니다. 이때 파일 경로가 존재하지 않으면 상위 디렉토리를 생성하고, 파일을 UTF-8로 기록합니다 [8]. 적용 후 `edits_safety.record_result()`를 호출하여 이번 시도가 성공했음을 기록합니다 [8].
⑤ 파일 쓰기 도중 예외가 발생하면 (예: 권한 문제 등) `record_result()`에 실패 결과를 기록하고 예외를 상위로 던집니다 [8]. 하나의 `apply` 명령으로 여러 파일을 처리할 수 있으므로, 루프를 돌며 모든 대상에 대해 이 절차를 반복한 뒤 완료됩니다.

**오류 처리 및 재시도:** `edits.apply` 과정에서 `should_apply`가 False를 리턴한 경우 (`ok_to_apply=False`) 해당 패치는 **스킵**되며 콘솔에 원인(reason)을 출력하고 넘어갑니다 [7]. 이로써 동일한 수정 요청이 이미 처리되었거나 반복 실패한 경우에는 실제 파일 쓰기 자체를 시도하지 않으므로 "실패->재시도->또 실패"의 **루프를 사전에 차단**합니다. 또한 `record_result`는 시도 결과를 지속적으로 누적 관리하므로, 이후 동일 패치 요청이 들어왔을 때 축적된 정보를 바탕으로 재시도 여부를 판단할 수 있습니다.

요약하면, `textops.py`는 **개별 파일에 대한 원자적 치환 작업과 기본 오류 처리** (줄바꿈 차이 허용 포함)를 제공하고, `edits_manager.py` + `edits_safety.py`는 **여러 수정의 배치 적용과 중복/오류 관리**를 담당합니다. `tasks.py`는 이러한 기능들을 CLI 명령으로 노출하며, 에이전트 명령이나 사용자의 CLI 입력에 따라 해당 함수를 호출합니다.

## 중복 수정 방지 전략 제안 (동일 요청 반복 차단)

**문제점:** GEMINI 에이전트 환경에서 **동일한 파일에 대한 동일 수정 요청**(예: 같은 `file_path`, `old_string`, `new_string` 조합의 치환 명령)이 단시간 내에 반복될 경우, 불필요한 실패가 누적되고 시스템 자원도 낭비됩니다. 특히 한번 실패한 수정이 아무 변화 없이 계속 시도되면, 사용자 입장에서도 혼란을 주고 에이전트의 신뢰도를 떨어뜨립니다.

**해결 전략:** 이러한 **중복 패치**를 예방하기 위해, 최근 시도한 수정의 식별자를 캐시에 저장하고 **중복 요청을 필터링**하는 메커니즘을 제안합니다. 구체적으로:
- **식별 키 설계:** 수정 시도의 고유 식별자로 파일 경로 + 수정 내용을 조합한 키를 사용합니다. 예를 들어 파일경로 + 구(old)->신(new) 문자열 쌍을 직렬화하여 키로 삼거나, 좀 더 정확하게는 **패치(diff)의 해시 값**을 키로 사용할 수 있습니다 [7]. 현재 구현에서도 `diff_hash = hashlib.sha256(diff.encode()).hexdigest()` 방식으로 고유 키를 생성하고 있습니다 [7].
- **시도 이력 저장:** 키에 해당하는 최근 시도 결과를 시간 정보와 함께 보존합니다. 본 레포지토리에서는 `.agents/edits_state.json` 파일을 이용해 영구적으로 상태를 저장하며, 각 키마다 마지막 시도시각(`last_ts`), 마지막 성공/실패 여부(`last_status`), 연속 실패 횟수(`fail_count`)를 기록합니다 [18] [10]. 이를 캐시로 활용하여 메모리뿐 아니라 다음 실행에도 누적 정보를 활용할 수 있습니다.
- **중복 판정 로직:** 새로운 수정 요청이 들어오면, 우선 해당 키로 이력 데이터베이스(혹은 캐시)를 조회합니다. 기록이 없으면 처음 시도로 간주하여 그대로 실행하고, 기록이 있는 경우 아래 정책을 따릅니다: - **최근 성공한 동일 패치**: 만약 동일 패치가 **최근에 성공적으로 적용**된 적이 있다면, 중복 시도로 간주해 이번 요청을 **무시**합니다. 기준 시간은 정책에 따라 결정되는데, 현재는 기본 60분 이내의 성공 내역이 있으면 "recently_applied" 사유로 스킵합니다 [9]. 이렇게 하면 이미 수정된 내용을 다시 수정하려는 불필요한 시도를 막을 수 있습니다.
- **연속 실패 중인 패치**: 만약 동일 패치가 **계속 실패**하고 있는 상황이라면, 일정 횟수 이상의 연속 실패가 누적된 경우 **일시적으로 재시도를 막습니다**. 현재 정책으로는 30분 이내에 3회 이상 실패한 패치에 대해 "backoff_due_to_failures" 사유로 적용을 스킵합니다 [9]. 이를 통해 반복 실패로 인한 소모를 줄이고, 원인을 파악하거나 환경이 바뀔 때까지 기다리도록 유도합니다.
- 위 두 조건에 해당하지 않으면 (성공 이력도 없고, 실패 누적 임계치도 아니면) **정상 실행**을 허용합니다. 이 판단 로직은 `edits_safety.should_apply()` 함수에 구현되어 있습니다 [9].
- **즉각 기록 업데이트:** 수정 적용 시도가 끝나면 (성공이든 실패든) 곧바로 결과를 기록/갱신하여 이후 판단에 반영합니다. `record_result()`는 호출 시마다 해당 키의 타임스탬프를 현재로 갱신하고, 성공 시 `last_status="success", fail_count=0`으로 초기화, 실패 시 `last_status="fail"`로 설정하고 `fail_count`를 +=1 증가시킵니다 [10]. 이러한 **실시간 기록**으로 다음 동일 요청 시 즉시 직전 결과를 고려하게 됩니다.

이러한 캐시/이력 기반 전략은 이미 레포지토리에 도입되어 있으며, 실제로 `edits_manager`의 `apply` 단계에서 이 로직을 활용하고 있습니다 (동일 패치 재적용 스킵 및 실패 누적 백오프) [19]. 앞으로 GEMINI 에이전트의 내부 로직에서도 이와 유사한 중복 방지 기능을 공유하면, 모델 전환이나 세션 간 맥락 손실로 인한 **"망각 반복 시도"** 문제도 완화할 수 있을 것입니다.

## 재시도 백오프 메커니즘 (실패 시 지연 전략)

**백오프(Backoff)**란 일정 횟수의 실패가 발생하면 즉각적인 재시도를 피하고 **시간 간격을 두어 재시도**하거나 아예 일정 시간 동안 시도를 막는 방법입니다. 본 문제에서는 동일 요청이 반복 실패하는 상황이 있었으므로, 다음과 같은 백오프 전략을 고려합니다:

- **임계조건 설정:** 몇 번의 연속 실패를 백오프의 트리거로 삼을지 결정해야 합니다. 현재 구현은 **3회 연속 실패**를 기준으로 하며 [19], 이 횟수는 조정 가능합니다. 너무 낮게 잡으면 일시적 오류에도 너무 오래 대기하게 되고, 너무 높게 잡으면 불필요한 시도가 누적될 수 있으므로, 환경에 맞춰 정하는 것이 중요합니다.

- **대기 시간(Window) 설정:** 백오프를 발동시키는 **시간 창(window)**과 **대기 지속 기간**을 설정합니다. 예컨대 "최근 30분 이내 3회 실패"를 조건으로 삼았다면, 30분 간격으로 판단합니다 [20]. 해당 조건에 걸리면 **일정 시간동안 (예: 30분)** 동일 패치를 시도하지 않도록 막습니다. 구현상으로는 최근 실패 시각과 횟수를 기록하여, **마지막 실패 이후 30분이 지나기 전까지는** `should_apply`가 False를 반환하게 됩니다 [9].
- **지연 방식:** 백오프가 활성화되면, 재시도를 **즉시 차단**할지, 아니면 **지연(schedule)**만 할지도 결정해야 합니다. 현재 `edits_safety.should_apply`는 단순 차단만 하고 지연 실행은 하지 않습니다 (사용자가 수동으로 다시 시도해야 함). 대안으로, 에이전트 내부에서 일정 시간 후 자동 재시도하는 로직을 둘 수도 있습니다. 그러나 이러한 자동화는 잘못 구현하면 또 다른 반복 시도를 낳을 수 있으므로 신중해야 합니다. 기본적으로는 **"지금은 실행시키지 않고 실패로 간주"**하는 식으로 일단 막고, 이후 사용자가 원하면 다시 명령하도록 유도하는 편이 안전합니다.
- **백오프 삽입 지점:** 백오프 판단 로직은 **실제 액션을 수행하기 직전 단계**에 넣는 것이 효과적입니다. 본 사례에서도 `cmd_apply` 내에서 파일을 write하기 전에 `should_apply`를 호출하여 백오프 여부를 결정하고 있습니다 [7]. 이렇듯 수정 도구를 실행하거나 파일에 변경을 가하기 바로 직전에 백오프 로직을 확인하면, 수행 자체를 스킵함으로써 추가적인 오류 발생을 원천 차단합니다. 또한, 백그라운드 스레드나 타이머를 이용해 **지연 재시도 예약**을 구현한다면, 실패 처리 직후 (예: `record_result`에서 fail_count 누적 후) 해당 타이머를 설정하는 방식을 고려할 수 있습니다. 예를 들어 연속 실패시 `time.sleep()`이나 스케줄링을 통해 일정 시간이 지난 후 재시도를 큐에 넣는 방식입니다. 그러나 이러한 기능은 LLM 에이전트가 관여하는 복잡한 환경에서는 예측이 어려우므로, 현재 수준에서는 **단순 차단**과 **로그 기록**만으로 충분히 효과를 보고 있습니다 [9].

정리하면, **백오프 전략**은 "일정 횟수 이상의 연속 실패가 발생하면, 설정된 기간 동안 동일 시도를 하지 않는다"로 요약됩니다. 이 로직은 이미 코드에 반영되어 (`3회/30분` 규칙) 있으며 [19], 필요시 구성값으로 노출하여 유연하게 조정할 수도 있을 것입니다.

## 통합 오류 보고 전략 (로그 중앙집중화)

현재 에이전트 및 여러 스크립트들이 각자 **성공/실패를 개별적으로 출력하거나 기록**하고 있어, 사용자가 전체 과정을 추적하기가 어렵습니다. 이를 개선하기 위한 통합 오류 보고 방안은 다음과 같습니다:

- **중앙 로그 DB/파일 활용:** 레포지토리에는 이미 `usage.db` (SQLite) 데이터베이스를 통해 명령 사용 내역과 에러를 저장하는 메커니즘이 있습니다 [14]. 하지만 모든 실패가 이 DB에 기록되지는 않으며(예를 들어 `check=False`로 실행된 경우), 일부 정보는 `edits_state.json`에, 또 일부는 콘솔 출력으로만 제공됩니다. **중앙집중식 로그**를 위해서는 하나의 통합된 경로로 모든 에러 정보를 보내야 합니다. 방법으로는:
- 모든 파일 수정 관련 함수가 **공통 로깅 유틸리티**를 호출하도록 합니다. 예를 들어 `edits_manager`에서 `record_result`를 할 때, 동시에 `usage.db`나 별도 로그 파일에 "패치 X 실패" 같은 이벤트를 추가 기록합니다. 또는 `runner.run_command` 내부에서 `check=False`인 경우에도 실패(stderr)에 특정 태그를 달아 DB에 넣도록 개선할 수 있습니다.
- 로그 대상은 단일 파일(JSON 또는 Markdown 등)로 할 수도 있습니다. 예를 들어 `.agents/edits_log.md`를 만들어, 각 수정 시도와 그 결과(성공/실패, 시간, 이유)를 append해나가는 것도 한 방법입니다. 이는 사람이 읽기 쉽고 버전관리에도 남겨둘 수 있다는 장점이 있습니다.
- **HUB.md 등 UI 통합:** 레포지토리에서 제공하는 HUB 시스템(문서)을 활용해 마지막 세션의 변경 파일 목록 등을 기록하듯이 [21], 실패 내역도 기록할 수 있습니다. 예를 들어 HUB.md의 **lastSession** 블록에 "Failures:" 섹션을 추가해 최근 실패한 수정 요청과 사유(예: file.py: old->new 교체 실패, 0 occurrences)를 요약할 수 있습니다. 이렇게 하면 사용자가 Active Tasks/HUB를 보며 무엇이 안됐는지 한눈에 파악하기 쉬워집니다.
- **에이전트 메시지와 연계:** GEMINI 에이전트 자체가 메시징 시스템을 갖추고 있으므로, 실패가 발생할 때 해당 에이전트의 inbox나 로그 채널로 노티를 보내는 방법도 있습니다. 예컨대 `broker.send`를 통해 agent="gemini"에게 타입="message"로 실패 정보를 담은 메모를 남기면 [22], 이후 에이전트가 이를 요약해 주거나 사용자가 `agent.inbox`로 확인할 수 있을 것입니다. 다만 이 접근은 구현 복잡도가 높아 우선순위가 낮을 수 있습니다.
- **요약 및 리포트 기능:** 최종적으로는 하나의 **리포트 생성기**를 두어, 분산된 로그들을 모아주는 것도 고려합니다. 예를 들어 `invoke prefs.show`로 현재 플래그들을 보여주듯이 [23], `invoke logs.failures` 등의

커맨드를 만들어 최근 실패 목록을 출력하도록 할 수 있습니다. 이는 실제 DB나 파일에서 fail 이벤트들을 쿼리하여 표시하면 됩니다.

결론적으로, **단일 진실 공급원(single source of truth)** 역할을 할 로그 시스템을 선정하고, 모든 관련 컴포넌트가 그 경로를 통해 기록을 남기도록 합의하는 것이 중요합니다. 현재는 `edits_state.json` + 콘솔 출력 + DB 등이 혼재되어 있으므로, **DB 기반으로 일원화**하거나, 또는 **HUB.md**와 같은 사용자 친화적 경로로 수렴시키는 방향을 제안합니다. 그렇게 하면 문제 발생 시 개발자나 사용자 모두 한 곳만 보면 되므로 디버깅이 수월해집니다.

## 잠재적 부작용 및 완화 방안

마지막으로, 새로운 중복 방지 및 백오프 로직 도입이 시스템에 미칠 **부작용**을 점검하고 대비책을 논의합니다:

- **(1) 정당한 수정 시도의 차단:** 중복 검사로 인해 혹시 필요했던 수정까지 막지는 않을까 하는 우려가 있습니다. 예를 들어 사용자가 의도를 가지고 같은 패치를 연달아 적용하려고 할 때, 시스템이 "최근에 적용되었다"면서 차단할 수 있습니다. 그러나 동일 파일에 동일 내용 패치를 반복 적용하는 것은 일반적으로 의미가 없으므로, 실질적인 문제는 크지 않을 것입니다. 만약 반드시 동일 패치를 재적용해야 하는 드문 케이스가 있다면, 임시로 캐시를 무시하거나 지울 수 있는 **우회 옵션**을 제공하면 됩니다. (예: `invoke prefs.set edits_enforce false`로 이 기능을 끄는 방법 [24]). 기본적으로는 해당 기능(`edits_enforce`)을 **True로 활성화**하되, 플래그 토글을 통해 사용자가 제어 가능하도록 한 점은 좋은 완화책입니다 [24].
- **(2) 에이전트의 인식 문제:** 에이전트(LM)가 동일한 지시를 반복했는데 시스템이 이를 조용히 스킵하면, 에이전트는 성공으로 착각하거나 계속 시도할 수 있습니다. 이를 완화하기 위해, **스킵했다는 사실과 이유를 에이전트에게 명시적으로 전달**해야 합니다. 현재 구현에서는 `edits.apply` 실행 시 `"skip <file>: <reason>"` 형태로 콘솔에 출력하고 있습니다 [7]. 만약 에이전트가 이 출력을 모니터링하거나 사용자에게 공유한다면, "이 패치는 방금 적용되었으니 생략한다"거나 "여러 차례 실패하여 잠시 중단한다"는 메시지를 인지할 수 있을 것입니다. 향후에는 이 메시지를 에이전트 대화창으로도 전달하는 방안을 고려해야 합니다.
- **(3) 성능 및 동시성:** 수정 이력 캐시를 조회하고 기록하는 작업은 JSON 파일 입출력을 수반하므로 약간의 성능 오버헤드가 있습니다. 그러나 파일 수정 빈도가 아주 높지 않고 JSON도 소용량이라 현재까지는 무시할만한 수준입니다. 다만 여러 에이전트가 동시에 수정 작업을 할 경우 동시 파일 접근 문제가 생길 수 있으므로, 필요하면 파일 잠금이나 원자적 연산을 적용해야 합니다. 현재 `_write_atomic` 등의 헬퍼가 구현되어 있어 원자적 기록을 돕고 있으며 [25], 충돌 가능성은 낮습니다.
- **(4) 기존 테스트/로직 영향:** 새로운 로직이 도입되면 기존 테스트 케이스나 워크플로우에 변화가 있을 수 있습니다. 예컨대 이전에는 5회 연속 시도하던 시나리오가 이제 3회 후 멈추게 된다면, 테스트는 이를 고려해 조정되어야 합니다. 레포지토리에서는 파일 에이전트 개선과 함께 테스트 케이스 작성을 계획하고 있으므로 [26], 새로운 중복 방지 기능도 테스트에 반영해야 합니다. 또한 다른 에이전트(Codex 등)가 이 로직을 공유할 경우 그쪽 흐름도 점검해야 합니다 (현재 `ALLOWED={"gemini","codex"}`로 다같이 쓸 수 있게 설계됨 [27]).
- **(5) 사용자 경험:** 사용자는 "내 명령이 무시되었다"는 것을 이해해야 합니다. 따라서 skip/backoff가 발동된 경우 **사용자에게 이유를 설명**하는 것이 중요합니다. 단순히 아무 반응 없거나 "성공"으로 잘못 표기되면 혼란이 생기므로, 앞서 언급한 통합 보고 채널이나 콘솔 출력에 친절한 메시지를 제공해야 합니다. 예를 들어 "[INFO] 최근 동일 수정이 실행되어 재적용을 건너뜁니다.", "[WARN] 해당 수정이 연속 3회 실패하여 30분간 재시도를 중단합니다." 등의 안내를 하면 사용자는 상황을 파악하고 다음 조치를 결정할 수 있을 것입니다.

**완화 요약:** 기본적으로 이번 개선은 **이중 안전장치**로 볼 수 있습니다. 에이전트 측에서도 유사 중복 방지를 구현하고, 레포지토리 측에서도 이를 강제함으로써 이슈 발생 가능성을 줄입니다. 혹시 모를 부작용은 설정을 통해 on/off할 수 있게 했고 [24], 충분한 로그와 메시지를 남겨 추적 가능성을 높였습니다. 앞으로도 실제 사용 시 피드백을 받아 **임계치 조정**이나 **세분화된 키 구성(예: 파일 경로 무시 여부)** 등을 다듬으면, 부작용은 최소화하면서도 문제 해결 효과는 극대화할 수 있을 것입니다.

**참고:** 해당 이슈는 2025-08-08자 task 로그에서도 언급되어 있듯이, 레포지토리 레벨의 완화책은 구현 완료된 상태이며 GEMINI 측의 추가 개선이 남아있는 상황입니다 [28]. 이번 분석을 통해 도출된 전략을 토대로 에이전트 쪽 코드에도

일관된 중복 방지 및 백오프 메커니즘을 적용하면, “실패한 동일 요청의 반복” 문제는 상당 부분 해소될 것으로 기대됩니다.

---

1 2 3 6 23 24 tasks.py
https://github.com/etloveaui/multi-agent-workspace/blob/6b523d7cfdfb4ce97c4e88f399e8d308103bef7d/tasks.py

4 5 11 12 13 textops.py
https://github.com/etloveaui/multi-agent-workspace/blob/6b523d7cfdfb4ce97c4e88f399e8d308103bef7d/scripts/textops.py

7 8 15 16 17 edits_manager.py
https://github.com/etloveaui/multi-agent-workspace/blob/6b523d7cfdfb4ce97c4e88f399e8d308103bef7d/scripts/edits_manager.py

9 10 18 20 edits_safety.py
https://github.com/etloveaui/multi-agent-workspace/blob/6b523d7cfdfb4ce97c4e88f399e8d308103bef7d/scripts/tools/edits_safety.py

14 runner.py
https://github.com/etloveaui/multi-agent-workspace/blob/6b523d7cfdfb4ce97c4e88f399e8d308103bef7d/scripts/runner.py

19 28 log.md
https://github.com/etloveaui/multi-agent-workspace/blob/6b523d7cfdfb4ce97c4e88f399e8d308103bef7d/docs/tasks/agent-repeated-modification-failures/log.md

21 hub_manager.py
https://github.com/etloveaui/multi-agent-workspace/blob/6b523d7cfdfb4ce97c4e88f399e8d308103bef7d/scripts/hub_manager.py

22 25 broker.py
https://github.com/etloveaui/multi-agent-workspace/blob/6b523d7cfdfb4ce97c4e88f399e8d308103bef7d/scripts/agents/broker.py

26 P1-2_File_Agent_Framework_Upgrade.md
https://github.com/etloveaui/multi-agent-workspace/blob/6b523d7cfdfb4ce97c4e88f399e8d308103bef7d/docs/proposals/P1-2_File_Agent_Framework_Upgrade.md

27 agent_manager.py
https://github.com/etloveaui/multi-agent-workspace/blob/6b523d7cfdfb4ce97c4e88f399e8d308103bef7d/scripts/agent_manager.py