ChatGPT

Scratchpad 디렉터리 지식 베이스 전환 방안

1. 최적의 분류 체계 (개선된 디렉터리 구조)

현재 `scratchpad`에는 업무 일지, 계획안, 디버깅 코드, LLM 대화 로그 등 다양한 종류의 파일들이 혼재되어 있습니다. 이를 체계적으로 정리하기 위해 제안된 5개 카테고리 – `1_daily_logs`, `2_proposals_and_plans`, `3_debug_and_tests`, `4_llm_io`, `_archive` – 는 전체 파일 유형을 대부분 포괄하고 있습니다 [1]. 각 카테고리는 다음과 같은 역할을 갖습니다:

- **1_daily_logs/** – 날짜별 **일일 작업 기록**을 보관합니다 (파일명에 날짜 형태인 YYYYMMDD 또는 "Daily_Log" 등을 포함하는 파일들). 예를 들어, `20250805_Daily_Log.md`와 같이 **날짜로 시작**하는 파일이나 `Daily_Log` 키워드를 포함한 파일은 여기 해당합니다 [2]. 또한 날짜로 된 태스크 목록(`20250731_TASK.md`와 같은 파일)도 일일 로그의 일환으로 간주해 함께 분류합니다. 필요에 따라 향후 **연도별 하위 디렉터리**를 만들어 로그를 관리할 수 있습니다 (예: `1_daily_logs/2025/`).

- **2_proposals_and_plans/** – 특정 프로젝트나 작업에 대한 **계획, 제안서, 아이디어** 등을 모읍니다. 이름에 "Plan", "Proposal", "Roadmap" 등의 단어가 들어가거나 [2], 문서 내용에 **목표**나 **단계 (Phase)**, **아이디어** 등이 서술된 파일들이 여기에 속합니다. 예를 들어 `Advance_Plan/` 디렉터리의 문서들, `P1-2_File_Agent_Framework_Upgrade/` 폴더 내의 설계 문서, 그리고 `Check.md` (검토 대기 중인 계획 초안) 등이 이 범주에 포함됩니다. 개발 관련 **청사진(blueprint)**이나 **설계 명세**(`...설계서.md`)도 모두 이 폴더로 정리합니다. (필요하다면 이 범주 내에서도 대형 프로젝트별로 하위 폴더를 둘 수 있지만, 현재는 파일 수가 많지 않아 모두 한 폴더에 두어도 무방합니다.)

- **3_debug_and_tests/** – **디버깅을 위한 임시 코드, 테스트 결과, 버그 분석 리포트** 등을 저장합니다 [3]. 파일명에 "debug", "test", "patch", "report" 등의 키워드가 있으면 이 범주로 분류합니다. 예를 들어, `debug_hub_stripper.py` (디버그용 임시 파이썬 코드), `ars-can-busoff-report.md` (버그 발생 보고서) 등이 해당됩니다. 또한 `p0_patch/`와 같이 **패치 코드 번들**을 담은 디렉터리나, 폴더 이름 또는 파일명에 `[P0]` 혹은 "Debug"가 포함된 내용(`Gemini-Self-Upgrade/[P0]Debug_6.md` 등의 **디버깅 기록**)도 이 폴더로 이동합니다. (기존에 특정 프로젝트 폴더 아래에 저장된 디버그 로그들은 프로젝트 context보다는 **자료 유형**에 따라 분류하는 것이 나중에 찾아보기 쉽기 때문에, 프로젝트와 별개로 해당 폴더로 모읍니다.)

- **4_llm_io/** – 기타 LLM (대화형 인공지능)과 주고받은 **프롬프트 및 응답 기록**을 보관합니다 [4] [2]. 예를 들어 `LLM_Requests/` 폴더와 `LLM_Answer/` 폴더에 들어있던 파일들이 이에 속하며, 파일명에 "LLM", "Prompt", "Request", "Answer" 등의 키워드를 포함하는 대화 로그, 프롬프트 설정, 응답 내용 파일을 이곳으로 옮깁니다. (기존의 `LLM_Requests`와 `LLM_Answer` 디렉터리는 이 `4_llm_io` 아래에 **하위 폴더**로 그대로 옮겨, 요청과 응답을 구분 유지할 수 있습니다.)

- **_archive/** – **더 이상 자주 사용되지 않지만 보관할 가치가 있는 파일**들을 모읍니다 [5]. 위의 어떤 분류에도 속하지 않는 파일들은 기본적으로 이곳으로 이동됩니다. 또한, 내용이 **구 outdated**되었거나 현재 프로젝트에 바로 필요하지 않은 과거 산출물 (예: 완료된 프로젝트의 문서, 오래된 메모 등)도 여기에 넣습니다. 예컨대, `README_run-gemini-cli.md`는 사용 방법을 설명하는 참조 문서로 추정되는데, 현재 진행 중인 작업과 직접 관련이 없으므로 archive 폴더로 보내 두는 것이 적절합니다. 향후 필요해질 **참고 문서나 옛 버전 파일**들은 `_archive`에서 찾을 수 있으며, `_archive`는 정렬 순서를 고려해 맨 뒤에 두기 위해 밑줄로 시작합니다.

▶ 카테고리 구조 개선 여부: 제안된 5개 카테고리는 전체적으로 파일들을 잘 포괄하며, **추가적인 상위 카테고리**는 현재로서는 필요 없어 보입니다. 다만, 몇 가지 **미세 조정**을 고려할 수 있습니다:

- **문서성 파일에 대한 처리:** `README_run-gemini-cli.md`와 같은 파일은 **참조 문서**로 볼 수 있는데, 별도 "documentation" 폴더를 만들기보다는 현 단계에서는 `_archive`에 포함시키거나, 해당 내용이 현재 진행 중인 계획과 관련 있다면 `2_proposals_and_plans`에 배치하는 방안을 고려했습니다. 하지만 이 파일은 실행 가이드 성격의 문서이므로, 우선 `_archive`에 보관하고 필요 시 `docs/`로 옮기는 것도 좋습니다.

- **일일 로그 세분화:** 현재 일일 로그 수가 많아지면 한 폴더에 너무 많은 파일이 쌓일 수 있습니다. 연도별 혹은 월별로 하위 폴더 (`1_daily_logs/2025/` 등)를 두는 방안은 **선택 사항**으로 남겨둡니다. 지금은 파일이 1년치 정도라 크게 문제 없지만, 향후를 대비한 구조를 열어둔 것입니다.

- **프로젝트별 임시 폴더 정리:** `Gemini_Start/`, `Gemini-Self-Upgrade/` 등 개별 폴더로 묶여 있던 자료는 위 카테고리별로 재분류합니다. 이 과정에서 해당 폴더 구조는 해체될 수 있습니다. 예를 들어 Gemini-Self-Upgrade 폴더 내에 **디버깅 기록만** 있다면 그 내용물은 `3_debug_and_tests`로 옮겨가고, 폴더는 빈 채로 남습니다. 이러한 빈 디렉터리는 최종적으로 정리 단계에서 삭제하거나 `_archive`에 폴더째 보관할 수 있습니다. 반면 P1-2_File_Agent_Framework_Upgrade/처럼 폴더 자체가 하나의 제안/계획 문서 모음이라면 폴더 전체를 `2_proposals_and_plans`로 이동하는 편이 더 자연스러울 것입니다. 즉, **폴더 단위로 맥락이 유지되는 경우** 해당 폴더째 이동하고, 폴더 내용이 여러 카테고리로 나뉠 경우 개별 파일 단위로 분류합니다.

이러한 개선된 분류 체계는 파일을 **용도와 성격별**로 구분하여, 개발자가 과거 자료를 찾거나 관리할 때 큰 도움이 될 것입니다.

2. 상세 분류 규칙 (파일 분류 Heuristics)

각 카테고리에 파일을 자동으로 분류하기 위한 **구체적인 규칙**을 정의합니다. 파일 이름 패턴, 파일 내용의 키워드, 파일 형식 등을 조합하여 **휴리스틱 규칙**을 설정합니다. 아래 표는 카테고리별 분류 기준을 요약한 것입니다:

카테고리	파일명 패턴 규칙	내용/키워드 규칙	파일 유형/기타
1_daily_logs	- `^\d{8}`로 시작 (예: `20250805_...` 형태) - 파일명에 `Daily_Log`, `daily`, `log` 등이 포함 (대소문자 무관) - 날짜 + `_TASK` 패턴 (예: `YYYYMMDD_TASK.md`)	- 문서 첫 부분에 해당 날짜 (예: `2025년 8월 5일`) 언급 - "작업 로그", "일일 보고" 등의 용어 포함 가능	- 주로 Markdown (`.md`) 파일 - 일자별 기록 표 형식이나 bullet 목록
2_proposals_and_plans	- 이름에 `Plan`, `Proposal`, `Roadmap`, `Idea` 포함 - 프로젝트 코드명/우선순위 패턴 (예: `P1-` or `[P1-2]`) 포함 - 기타 기획 관련 키워드: `Design`, `Spec`, `Blueprint` 등	- 문서에 "목표", "계획", "단계", "발안" 등의 단어 빈도 높음 - 구조상 **개요-세부 단계-결론** 형식 (제안서 형식) - 제목에 "계획서", "제안서" 등 명시	- 주로 Markdown이나 문서 파일 (`.md`, `.txt`, `.docx` 등) - 폴더 전체가 기획 문서 모음인 경우 폴더 단위 이동

카테고리	파일명 패턴 규칙	내용/키워드 규칙	파일 유형/기타
3_debug_and_tests	- 파일/폴더명에 `debug`, `Debug`, `test`, `patch` 포함 - 이름에 `report`, `issue`, `error` 등의 문제 해결 맥락 단어 포함 - 예: `*_report.md`, `debug_*.py`, `[P0]Debug_*` 등	- 내용에 에러 로그, **스택 트레이스** (traceback) 또는 "Error:", "Exception" 등의 키워드 - 테스트 결과 요약 (예: "Assertions: 10 passed") - 버그 원인 분석 내용 포함 가능	- **코드 파일** (`.py`, `.sh` 등 임시 스크립트) - **로그/결과 파일** (`.md`, `.log`, `.txt`) - 디렉터리 (예: 패치 모음 폴더)도 포함
4_llm_io	- `LLM` 관련 문자열 포함: 예 `LLM_`, `Prompt`, `Request`, `Answer`, `Response` 등 - LLM 입출력 폴더명 (`LLM_Requests`, `LLM_Answer`)	- 프롬프트와 응답 형식의 대화 내용 포함 - 예: 내용에 "User:", "Assistant:" 식의 문구 - 시스템/모델 응답 기록 ("``yaml`", JSON 등 포맷 포함 가능) | - 주로 `.md` 대화 로그 또는 `.json` 등 - 기존 LLM 관련 폴더는 이 폴더로 이동 (구조 유지)	
_archive	- **다른 규칙에 해당되지 않는** 모든 파일/폴더 - 파일명이 `old`, `backup` 등을 포함 (과거 백업) - 오래된 양식 또는 특수한 형식 파일	- (특정 내용 패턴보다는 **활용 빈도**에 근거) - 내용상 최신 맥락과 동떨어진 문서 (예: 지난 분기 계획, 구버전 매뉴얼 등)	- 모든 형식 (예: PDF, 이미지 등도 보관) - 최근 6개월 이상 열람/수정 되지 않은 파일

규칙 적용 우선순위: 분류는 **위에서부터 차례로** 적용합니다. 즉, 일일 로그 패턴에 해당하면 바로 `1_daily_logs`로 분류하고, 그렇지 않으면 다음 범주의 규칙을 검사하는 방식입니다. 한 파일이 여러 규칙에 중복으로 걸리는 경우를 최소화하기 위해 패턴들을 설계했습니다 (일반적으로 파일명/내용이 **명확히 하나의 범주**에 들어맞도록 작명되었음). 그래도 모호한 경우가 있다면, 예를 들어 **날짜 패턴**과 **특정 계획 키워드**를 모두 포함한 파일이라면, **보다 구체적인 범주**에 넣는 것을 원칙으로 합니다. (날짜로 시작하지만 내용이 특정 계획 제안서 형태인 경우 `2_proposals_and_plans`로 분류 등.)

예시 검증: 위 규칙을 현재 파일 목록에 적용하면 다음과 같습니다.

- `20250805_Daily_Log.md` – 파일명이 `2025...`로 시작하고 `Daily_Log`를 포함하므로 **1_daily_logs**로 분류 [2].
- `P1-2_File_Agent_Framework_Upgrade/제미나이 CLI 작업 지시서 (P1-2 실행).md` – 파일 경로와 이름에 `P1-2` 및 "작업 지시서" (계획 문서)이 있으므로 **2_proposals_and_plans**에 속함.
- `debug_hub_stripper.py` – 파일명에 `debug` 포함, `.py` 스크립트이므로 **3_debug_and_tests**로 이동.

- `Gemini-Self-Upgrade/[P0]Debug_6.md` – 파일명에 `Debug` 있고 디버그 시도 기록이므로 **3_debug_and_tests**로 이동 (기존 폴더에서 분리).
- `LLM_Requests/2025-08-01_user_prompt.txt` – 경로에 `LLM_Requests`가 있으므로 **4_llm_io**로 이동 (폴더째 이동하거나 통째로 포함).
- `README_run-gemini-cli.md` – 다른 어떤 규칙에도 맞지 않으므로 **_archive**로 보관 (추후 참고자료).

위와 같은 **판단 규칙 표**를 코드로 구현하여 각 파일의 이동 대상 디렉터리를 결정하게 됩니다.

3. 자동화 작업 지시서 (`invoke organize-scratchpad` 명령)

마지막으로, 위 분류 체계와 규칙에 따라 `scratchpad`를 자동 정리하는 명령 (`invoke organize-scratchpad`)의 **구현 방안**을 제시합니다. 이 명령은 다음 절차로 동작합니다 [6]:

1. **파일 분석 단계:** `scratchpad` 디렉터리를 **재귀적으로 스캔**하여 모든 파일 및 폴더 경로를 수집합니다. 각 항목에 대해 앞서 정의한 휴리스틱을 적용하여 **대상 카테고리 디렉터리**를 결정합니다 [7]. 이때 이미 정리된 폴더 (예: `scratchpad/1_daily_logs/` 등이 존재한다면) 내부의 파일은 제외하거나 무시하여, 이 명령을 반복 실행해도 중복 이동이 발생하지 않도록 합니다. 디렉터리의 경우, 이름 자체가 규칙에 부합하면 폴더 단위로 분류하고, 그렇지 않으면 내부 파일들을 개별 평가합니다.

2. **이동 계획 생성:** 각 파일/폴더의 **원본 경로**와 **목표 경로** 쌍의 리스트를 생성합니다 [7]. 이 "이동 계획 (Move Plan)"에는 예컨대 `(scratchpad/Gemini_Start/A.md, scratchpad/2_proposals_and_plans/A.md)` 같은 항목들이 포함됩니다. 아직 실제로 옮기지는 않고, 계획만 세웁니다. 만약 어떤 파일이 **규칙상 두 범주에 모두 해당**하여 모호함이 있다면, 우선순위에 따라 하나로 결정하되, 이러한 결정을 로그에 남겨 **사용자가 사전에 인지**할 수 있게 합니다.

3. **사용자 확인 단계:** 생성된 이동 계획을 터미널에 출력하여 **사용자 검토를 받습니다** [8]. 이 때 Python의 `rich` 라이브러리 등을 활용하여 표 형태로 보기 좋게 출력합니다. 예를 들어, `rich.table.Table`을 사용해 Source (현재 경로)와 Destination (이동 경로) 열을 가진 표를 만들고, 각 파일을 한 행으로 표시합니다. 표 머리글에는 대상 카테고리도 함께 명시하여 한눈에 분류 결과를 파악할 수 있게 합니다. 사용자는 이 표를 보고 이동 계획을 확인한 뒤, **승인 여부를 선택**하게 됩니다 (예: `Proceed with these changes? [y/N]` 프롬프트 출력).

4. **승인 시 실행 단계:** 사용자가 확인하여 승인을 하면 실제 **파일 이동을 수행**합니다 [9]. Python 코드를 작성한다면 `shutil.move` (또는 Path 객체의 `rename`)를 사용하여 파일을 옮깁니다. 이 단계에서 다음 사항을 특별히 처리합니다:

5. **디렉터리 생성:** 대상 경로에 해당 카테고리 폴더가 없으면 `os.makedirs` 등을 통해 미리 생성합니다 (예: `scratchpad/1_daily_logs/` 폴더가 없다면 생성).
6. **이름 충돌 처리:** 만약 동일한 이름의 파일이 대상 위치에 이미 존재하면 **파일명에 접미어**를 붙여 충돌을 피합니다 [9]. 예를 들어 `example.md`가 이미 있다면 `example_1.md`처럼 숫자를 붙입니다. 새로운 이름도 충돌하면 번호를 증가시켜 (`_2`, `_3` …) 유일한 이름을 찾습니다. (원한다면 덮어쓰기 전에 사용자에게 재확인 받을 수 있지만, 자동화 편의를 위해서는 숫자 붙여 별도 보관이 안전합니다.)
7. **폴더 단위 이동:** 폴더 자체를 옮기는 항목은 `shutil.move`를 통해 **디렉터리째 이동**합니다. 이미 대상 폴더에 동일 이름 폴더가 있을 경우 위와 같은 방식으로 이름을 바꾸거나, 병합이 애매하면 `_1` 등을 붙여 폴더명을 변경합니다.
8. **빈 폴더 정리:** 개별 파일들을 모두 이동하고 나서 원본 폴더가 비게 되면, 해당 **빈 디렉터리를 삭제**합니다 (`os.rmdir` 이용). 단, 혹시 남겨둘 필요가 있는 폴더(예: `_archive`로 이동한 폴더)는 삭제하지 않습니다.

9. **예외 처리:** 이동 중 오류가 발생하면 해당 작업을 건너뛰고 오류 메시지를 기록하지만, 나머지 파일 이동은 이어서 진행합니다. (원자적 이동이 필요하면 전체 승인 후 한꺼번에 이동하는 것도 고려 가능하지만, 여기서는 파일 단위로 순차 이동하되 오류시 로그 남김.)

10. **로그 및 완료 보고:** 이동이 모두 완료되면 콘솔에 요약을 출력하고, 별도의 **로그 파일** (예: `scratchpad/organize_log.txt`)을 남겨 어떤 파일을 어디로 옮겼는지 기록합니다. 이 로그에는 타임스탬프와 함께 **승인자 확인** 표시도 포함하여, 나중에 누가 언제 정리했는지 추적할 수 있게 합니다.

위 과정을 의사 코드(Pseudocode) 형태로 표현하면 다음과 같습니다:

```
@task
def organize_scratchpad(c):
    """scratchpad 디렉터리를 정리하는 Invoke task"""
    base_dir = Path("scratchpad")
    move_plan = []  # (src, dest) tuples

    # 1. 파일 및 폴더 분석
    for item in base_dir.iterdir():
        if item.name in ["1_daily_logs", "2_proposals_and_plans",
                         "3_debug_and_tests", "4_llm_io", "_archive"]:
            continue  # 이미 정리된 폴더는 스킵
        if item.is_dir():
            # 폴더 전체를 분류할지, 내부 파일을 분류할지 결정
            target_category = determine_category(item, is_dir=True)
            if target_category is not None:
                # 폴더 자체를 이동
                dest_dir = base_dir / target_category / item.name
                move_plan.append((item, dest_dir))
                continue
            else:
                # 폴더 내부 개별 파일 분류
                for file in item.rglob('*'):
                    if file.is_file():
                        cat = determine_category(file)
                        dest_path = base_dir / cat / file.name
                        move_plan.append((file, dest_path))
        else:
            # 개별 파일 분류
            cat = determine_category(item)
            dest_path = base_dir / cat / item.name
            move_plan.append((item, dest_path))

    # 2. 이동 계획 생성 완료 -> rich 테이블로 출력
    table = Table(title="Scratchpad Organization Plan")
    table.add_column("Source", style="dim")
    table.add_column("Destination", style="cyan")
    table.add_column("Category", style="magenta")
    for src, dest in move_plan:
        category_name = dest.parts[-2]  # 예: .../3_debug_and_tests/filename.md ->
"3_debug_and_tests"
```

```
            table.add_row(str(src), str(dest), category_name)
    console.print(table)

    # 사용자 확인
    confirmation = console.input("\n위 계획대로 파일을 이동할까요? [y/N]: ")
    if confirmation.strip().lower() != 'y':
        console.print("사용자 취소 - 정리 작업을 종료합니다.")
        return  # 승인되지 않으면 종료

    # 3. 승인된 경우 실제 이동 실행
    for src, dest in move_plan:
        dest.parent.mkdir(parents=True, exist_ok=True)  # 대상 폴더 생성
        if dest.exists():
            # 파일 이름 충돌 처리: _1, _2 등 붙이기
            base_name = dest.stem
            ext = dest.suffix
            i = 1
            new_dest = dest
            while new_dest.exists():
                new_dest = dest.parent / f"{base_name}_{i}{ext}"
                i += 1
            dest = new_dest
        # 파일/폴더 이동 수행
        shutil.move(str(src), str(dest))
        console.print(f"[green]Moved:[/green] {src} -> {dest}")

    console.print(f"\n정리 완료! 총 {len(move_plan)}개의 항목을 이동했습니다.")
    # (선택) 로그 파일에 기록
    with open(base_dir / "organize_log.txt", "a", encoding="utf-8") as logf:
        logf.write(f"[{datetime.now()}] Moved {len(move_plan)} items.\n")
        for src, dest in move_plan:
            logf.write(f"  {src} -> {dest}\n")
```

(위 코드는 이해를 돕기 위한 의사 코드이며, 실제 코드 구현 시 파일 경로 처리나 예외 처리를 더 면밀히 다듬어야 합니다.)

이렇게 구현된 `invoke organize-scratchpad` 명령을 실행하면, 먼저 AI가 **분류 제안**을 하고 인간이 **검토 및 승인**한 후에 최종적으로 **파일 이동을 실행**하게 됩니다 [6]. 이는 사람의 판단을 한 번 거치므로 잘못된 자동 분류로 인한 사고를 예방할 수 있으며, 모든 이동 내역이 기록으로 남아 투명한 관리가 가능합니다.

궁극적으로 이 자동화 스크립트는 개발팀의 지식 관리 효율성을 높이고, **scratchpad 디렉터리**를 유의미한 **지식 베이스**로 탈바꿈하는 데 기여할 것입니다.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] Request_for_P2-UX_Scratchpad_Organization.md
file://file-FPuHZhkGTeR6W3G9GcKUWG