ChatGPT

Spring Cleaning Analysis – *concrete-mix-database* (Branch: database-refresh)

Overview

The **database-refresh** branch of the *concrete-mix-database* repository has accumulated a large number of standalone Python scripts (over 50 at the project root). These scripts – often named with prefixes like `import_*.py`, `validate_*.py`, `check_*.py`, `test_*.py`, `fix_*.py`, and `cleanup_*.py` – were created during development for data import, validation, one-off fixes, and analyses. This report examines these scripts and provides recommendations to declutter and organize them without affecting critical web app components. We also review key documentation files for clarity and consistency, and identify code quality improvement opportunities across the scripts. All critical application directories and files (e.g. `concrete_mix_project/`, `cdb_app/`, `refresh_status/`, `templates/`, `static/`, and `manage.py`) will remain intact as instructed. The goal is a cleaner repository structure where essential tools are organized logically and legacy/experimental scripts are clearly segregated.

Current State Summary: The root folder contains numerous Python scripts created over time for tasks like data ETL, database integrity checks, debugging, and data fixes. Many of these were ad-hoc solutions (especially for the DS6 dataset import issues) that have since been superseded by more robust approaches [1] [2]. Important improvements (such as the DS6 “material reference system” for imports) have rendered several older scripts obsolete [3]. However, most of these files still reside at the repository root, making it difficult to discern which are actively useful and which are legacy.

Approach: We audited the root-level scripts, grouping them by purpose, and assessed each for usage and relevance. The recommendations favor **conservative cleanup** – when in doubt, we suggest **archiving rather than outright deletion** to avoid losing potentially useful logic. We propose a new `scripts/` directory hierarchy to organize the scripts by category, and we highlight documentation adjustments and code quality improvements to support this cleanup. Each recommendation below includes the proposed action, justification, and an assessment of risk (low/medium/high) to the project’s stability. A summary table of all file-specific actions is provided at the end of the report for quick reference.

Root-Level Python Scripts Audit

Import/ETL Scripts

These scripts handle data loading or transformation (often related to external *“DS” datasets). In the current branch, the primary import logic has been consolidated into the Django management command* `import_ds.py`* (under `cdb_app/management/commands/`) for dataset imports [3]. Earlier, developers used ad-hoc scripts to ingest datasets and mapping files:

- `create_corrected_mapping.py` – Generates properly formatted column mapping files for dataset import (developed during the DS6 fix) [4]. **Action:** *Move to* `scripts/etl/`.

- *Justification:* It's a useful ETL tool for preparing mapping CSVs. Keeping it in an `etl` folder reflects its role in data import. This script was crucial for DS6 and may be reused or adapted for future datasets [5].
- *Risk:* **Low.** It's a standalone utility. Moving it has no impact on the web app.

- **Dataset-specific import scripts** (e.g., any `import_<dataset>.py` if they exist). *No dedicated dataset import scripts were found in this branch,* likely because `import_ds.py` (a managed command) supersedes them. If any legacy import scripts (for earlier DS1–DS5 datasets or CSV ingestion) exist at root, they should be **archived**.

- *Justification:* Redundant import scripts would be outdated – the unified `import_ds.py` covers their functionality under Django's context. Consolidating to one import pipeline reduces confusion.
- *Risk:* **Low.** These legacy scripts are not used by the app (and new imports should go through the managed command). Archiving retains the logic just in case, without cluttering the root.

- `manage_cdb.py` – A utility to run Django commands against the "cdb" database specifically [6] [7]. **Action:** *Archive or move to* `scripts/active/` (pending decision on its necessity).

- *Justification:* This script was helpful when the project had multiple database schemas (original vs new). Now that the app is unified on a single *"cdb"* database (as of v1.0.1) [8] [9], Django's standard `manage.py` likely suffices. The custom logic in `manage_cdb.py` (e.g. forcing `--database=cdb`) may no longer be needed after consolidating to one database in settings [10].
- *Risk:* **Low** (if indeed obsolete) or **Medium** (if some team workflows still rely on it). To be safe, we do not delete it outright. Archiving it (or placing in `scripts/active/` with a note) ensures it's available if needed for edge cases, but it's no longer in the way for everyday tasks.

- `run_server.py` – A development server launcher that printed banners for dual-app mode [11]. **Action:** *Archive* this script.

- *Justification:* It refers to an "Original Concrete Mix App" vs "New CDB App" on different URL paths [11], reflecting an older development phase with two parallel apps. Since the old `concrete_mix_app` was retired (single-app architecture adopted) [12] [13], this dual-run script is outdated. Developers can simply use `python manage.py runserver` now.
- *Risk:* **Low.** This is a convenience script not required for the application. Archiving it removes potential confusion (someone might try to use it not realizing the old app is gone) while preserving it in case it's ever needed for reference.

Validation & Check Scripts

These scripts were used to verify data integrity and evaluate dataset imports. Many were introduced during troubleshooting of dataset DS6 and may overlap in purpose with newer "health check" tools:

- `ds6_health_check.py` – Checks integrity of the DS6 dataset import (producing counts of mixes, linked materials, etc.) [14]. **Action:** *Move to* `scripts/validation/`.
- *Justification:* It remains a valuable validation tool for ensuring data quality post-import. Placing it in a dedicated validation folder signals its purpose. It could potentially be generalized for other datasets in the future.

- *Risk:* **Low.** This script does not interact with the running app; it reads database state to report on data consistency. Moving it has no side effects on production.

- `generate_ds6_health_flags.py` – Generates a CSV of "health flags" or quality metrics for each mix in DS6 [14]. **Action:** *Move to* `scripts/analysis/` (or `scripts/validation/` – depending on whether we treat it as an analysis output tool).

- *Justification:* This script analyzes data completeness (e.g., missing strength data) and outputs results, which is part validation and part analysis. It was created to assess DS6 data quality after import [15]. Storing it with other analysis/validation scripts keeps it accessible for similar assessments (and keeps the root clean).

- *Risk:* **Low.** It's an offline analysis tool. No impact on the web app by relocating it.

- `check_mix_codes.py`, `check_database.py`, `find_all_mixes.py` – Quick diagnostics written during data troubleshooting. For example, these were used to identify inconsistent mix naming and to locate all mixes related to DS6 in the DB [16] [17]. They were deemed *temporary* and in fact were removed once the DS6 issues were resolved [1]. **Action:** *Do not reintroduce; keep as archived references if they still exist in the branch.*

- *Justification:* These served their purpose during the DS6 import fiasco but are no longer needed after implementing the correct import and purge process [1]. At most, retain them in `scripts/archived/` for historical reference. However, if they've already been deleted from the branch as the DS6 changelog suggests, then nothing needs to be done (just ensure no similar redundant scripts linger).

- *Risk:* **Low.** They are obsolete. Restoring or using them would risk confusion and potential misuse. Archiving (or keeping them deleted) has no downside.

- `validate_property_linking.py` – Another DS6-era script that attempted to verify material-property links after import [17]. **Action:** *Archive.*

- *Justification:* It was part of an unsuccessful approach superseded by the new reference-key import system [18] and health check scripts. The DS6 changelog explicitly lists it among removed scripts [1]. Any logic still relevant is likely incorporated into the current import or validation scripts.
- *Risk:* **Low.** Not needed for current workflows.

Database Management / Fix Scripts

This category includes scripts that directly modify database records or schema to fix issues. They are powerful and must be handled carefully. In our cleanup, we isolate them in a `scripts/database_management/` folder and clearly mark those that are deprecated:

- `thorough_ds6_purge.py` – A comprehensive purge script that wiped all DS6-related data from the database [19] [20]. **Action:** *Move to* `scripts/database_management/` and **label as archival/for reference only**.
- *Justification:* This script was the final solution to clean out corrupted DS6 data so a fresh import could be done [20]. The branch documentation indicates this script was **kept for reference** after use [21]. It's wise to retain it (in case a similar purge is needed or for audit purposes), but it

should live in an `archived` or `database_management` folder to signal it's not to be run casually.

- *Risk:* **Low.** Moving it doesn't affect runtime, and since it's not needed unless re-running a purge, there's minimal risk. (Accidentally executing it in production would be catastrophic, but relocating it and documenting its purpose mitigates that risk.)

- `fix_ds6_final.py` – A last attempt at an all-in-one DS6 fix via script (likely updating mix codes or links) that was eventually abandoned in favor of other solutions [22] [1]. **Action:** *Archive.*

- *Justification:* Listed as a removed temporary script [1]. Its intended fixes were implemented differently (via SQL or the import reference system). Keeping it in `scripts/archived/` preserves the logic for posterity, but it has no active role now.

- *Risk:* **Low.** Not used by any active process.

- `link_material_properties.py` – Script to link material properties to mixes post-import (another DS6 troubleshooting tool) [17]. **Action:** *Archive.*

- *Justification:* Also flagged as removed [1]. The approach it took was flawed and replaced by the material reference system [23] [24]. No current need to run it.

- *Risk:* **Low.** Obsolete logic.

- **Other data-fix scripts:** If there are any other Python scripts for data fixes (e.g., for other datasets or general cleanup) – for instance, something like `cleanup_orphan_records.py` or `standardize_mix_codes.py` – they should be reviewed similarly. The branch's **MASTER_CHANGELOG.md** indicates some fixes were done via direct SQL scripts (e.g. `fix_ds6_mixcodes.sql`, `fix_ds6_ratios.sql`, `cleanup_materials.sql`) [25] [26] rather than Python, so we may not have many Python-based fix scripts beyond DS6. Any that do exist should be moved to `scripts/database_management/` if still relevant, or to `scripts/archived/` if they correspond to one-off data corrections that won't be run again.

- *Justification:* Keeping all DB-altering scripts in one place helps with oversight. For example, if there was a script to merge duplicate materials or update records, it belongs with database management tools. If such a script's function has been achieved through migrations or SQL, it can be archived to avoid accidental reuse.
- *Risk:* **Low** (for archiving) to **Medium** (if we misidentify a script as outdated when it's still needed). We will err on the side of caution by archiving rather than deleting any borderline cases.

Analysis Scripts

A few scripts appear to have been used for analysis or reporting on the data rather than modifying it. These often produce outputs saved in the repository (e.g., in the `analysis_output/` folder):

- **Statistical analysis scripts** – For example, an OLS regression was performed on the dataset (the output `analysis_output/ols_baseline.txt` is checked into the repo) [27]. There might be a script (or notebook) that generated this baseline model. If such a script exists (e.g., `ols_baseline.py` or similar), it should be moved to `scripts/analysis/`.

- *Justification:* Analysis scripts are not part of the core app; they are exploratory or reporting tools. Organizing them under `scripts/analysis/` keeps them accessible to data scientists or developers without cluttering the root. It also signals that these are not required for running the web application.

- *Risk:* **Low.** These do not affect application logic. If anything, moving them clarifies that they are optional tools.

- **Performance or data exploration scripts** – e.g., any `test_*.py` used to test performance of queries or simulate API usage. (We did not find specific file names in this category via the repository search, but the inclusion of the `test_*.py` pattern in the prompt suggests such scripts exist.) **Action:** *Move to* `scripts/experimental/` or `scripts/analysis/` depending on their purpose.

- *Justification:* If a script was used to test something (like an API response or an algorithm), it's likely experimental or for developer insight. Placing it in `experimental` indicates it's not an official tool but rather a throwaway experiment (unless it's more of a formal analysis, then it goes to analysis).

- *Risk:* **Low.** If it's truly a test script, it's not critical. We just ensure it's not mistaken for a production tool.

- **Visualization or reporting scripts** – (If any exist; not explicitly seen in our audit). If the project has scripts for generating charts or reports from the data (for internal reporting), those belong in `scripts/analysis/` as well. They should be kept separate from validation scripts – analysis focuses on deriving insights, whereas validation checks data integrity.

Experimental / Legacy Scripts

Finally, any script that doesn't clearly fit the above categories or is known to be a one-off experiment should go to `scripts/experimental/` or `scripts/archived/`:

- **Experimental prototypes:** If there are scripts that trial new features or approaches (e.g., trying a new import strategy or a different library usage) which were never fully adopted, mark them as experimental. For example, if someone wrote a `import_v2.py` or a script to test a new ORM query, those should be moved to `scripts/experimental/` for now.
- *Justification:* This signals to future maintainers that these scripts are not part of the normal toolkit. It allows keeping the code for reference without causing confusion.

- *Risk:* **Low.** By definition these were not production-critical.

- **Archived (deprecated) scripts:** As covered above, many DS6-related fix scripts are deprecated. We will collect all such clearly unused scripts in `scripts/archived/`. This includes all files explicitly listed as "removed temporary scripts" in the DS6 changelog (check_mix_codes, find_all_mixes, check_database, fix_ds6_final, link_material_properties, validate_property_linking, purge_ds6_data) [1], and any similar older scripts for other datasets or tasks.

- *Justification:* Archiving them preserves institutional knowledge (in case we ever need to review how a past issue was handled) while entirely removing them from the active code paths. We avoid outright deletion as a safeguard in case our understanding of their usefulness is incomplete.

- *Risk:* **Low.** They won't be invoked accidentally from an `archived` folder and no app code depends on them. The main risk would be a future contributor not realizing an archived script exists and duplicating effort – but that's mitigated by keeping an index (like this report or an updated README section) of what's in `archived/`.

Note: The `manage.py` file (Django's default manage script) remains at the project root unchanged, as it is essential for normal operations. We do **not** move or rename `manage.py` (doing so would break the Django workflow). Likewise, core application packages (`concrete_mix_project/`, `cdb_app/`) and web assets (`templates/`, `static/`) remain untouched by this cleanup. Only the miscellaneous utility scripts are being reorganized.

Proposed `scripts/` Folder Structure

To impose order on the script sprawl, we propose creating a top-level `scripts/` directory with clearly named subfolders. The following structure and content assignments are recommended:

- `scripts/active/` – **High-value, actively used scripts or utilities.** This includes any general management tools still in regular use. For instance, if we decide to keep `manage_cdb.py` accessible (assuming some workflows might still need it), we could place it here. Currently, *ideally this folder may even be empty* (if no script is routinely used in production or development beyond what manage.py covers). Another candidate: if there's a script to run routine refresh jobs or combine multiple steps (none identified explicitly in audit, but if "refresh_status" relates to a script, it would go here).

- *Rationale:* Separating truly "active" scripts ensures that anything in this folder is maintained and tested. It gives developers a quick sense of which tools are current.

- `scripts/etl/` – **Data import and ETL scripts.** All scripts related to extracting, transforming, or loading data into the database go here. Confirmed examples: `create_corrected_mapping.py`, any one-time import scripts, or data transformation helpers. If a script interacts with CSV/Excel files for import or sets up mapping structures (like `create_column_map.py` found in the `etl/temp/` folder in the repo), it belongs here.

- *Rationale:* During development, ETL scripts were scattered (some in root, some in an `etl` subfolder). Consolidating them under `scripts/etl/` makes it easier to locate data-import logic. It also aligns with how the project already has an `etl/` data folder – now the code that *performs* ETL is grouped, complementing the data files.

- `scripts/validation/` – **Data validation and integrity check scripts.** This includes `ds6_health_check.py`, any script that performs consistency checks on the database (like ensuring naming standards, detecting duplicates, verifying foreign key linkages, etc.). If a script primarily reads data and reports issues without altering anything, it should live here.

- *Rationale:* By grouping these, one can run a suite of validation scripts after data import or periodically to monitor data health. It emphasizes their read-only/checking nature, separate from scripts that *change* data.

- `scripts/database_management/` – **Database maintenance and fix scripts.** Scripts that directly manipulate the database for cleanup or fixes go here. Examples: `thorough_ds6_purge.py`, any future script to batch-update records or perform migrations outside of Django's migration system. Also, if any raw SQL files are kept in the repo for manual execution (like the `.sql` fixes in the changelog), consider moving them into a `database_management` subfolder (or a dedicated `sql/` folder) for organization.

- *Rationale:* These scripts often need extra caution. Having them in one place allows adding README notes or warnings in that directory (e.g., "**CAUTION:** these scripts will modify production data – review before running**"). It helps prevent running them unintentionally.

- `scripts/analysis/` – **Data analysis and reporting scripts.** Any code that produces analytical insights, reports, or visualizations from the data, without being part of the app's features, belongs here. Our audit identified at least the OLS regression (implied by the saved output) – presumably the code that generated it would reside here. Also, `generate_ds6_health_flags.py` could be placed here if we classify its output (a CSV of flags) as an analysis artifact rather than a pure validation (it's somewhat both).

- *Rationale:* Separating analysis ensures that exploratory code (often using pandas, numpy, etc.) is not mixed with core app code. It can have its own dependencies or environments if needed. This also invites contributors to add notebooks or analysis scripts in an organized way, which is useful for data-driven projects.

- `scripts/experimental/` – **In-progress or experimental scripts.** Anything not fully validated or only used for trial runs goes here. This might include test scripts (`test_*.py`) used to prototype or debug, as well as partially implemented utilities that were never integrated.

- *Rationale:* This is essentially a sandbox area within the repo – code here is not assumed to be maintained or used, just kept for reference. Over time, items here can be promoted (to active/etl/etc.) or archived/deleted as their status changes.

- `scripts/archived/` – **Deprecated or superseded scripts.** All scripts known to be outdated, duplicated, or no longer applicable should be moved here (with a README note stating they are retained for historical reference only). The DS6 workaround scripts fall in this category (as per DS6 changelog) [1]. We would populate this folder with: `check_mix_codes.py`, `find_all_mixes.py`, `check_database.py`, `fix_ds6_final.py`, `link_material_properties.py`, `validate_property_linking.py`, `purge_ds6_data.py`, `run_server.py` (since it's outdated), and possibly `manage_cdb.py` if we determine it's fully obsolete.

- *Rationale:* This collects all "do not use" scripts in one place, clearly separated from active code. It prevents accidental use of old scripts and declutters the root, while still version-controlling the files in case we need to look up old logic. Each archived script can have a header comment inserted (if not already) stating it's deprecated.

This new structure addresses the immediate clutter and establishes a convention for any new scripts the team creates. It will be important to also update any documentation (e.g., the README or developer guide) to instruct developers to use the new script locations. For instance, if the README currently tells users to run `python some_script.py`, it should be changed to `python scripts/etl/some_script.py` as appropriate. We should also verify that moving these scripts does not break any relative imports or file path assumptions inside them (most of these are standalone and likely refer to project paths or Django settings via environment variables, which should still work if executed from the repository root).

Documentation Files Review

A thorough documentation review was conducted on the following key files: **README.md**, **CHANGELOG.md**, **MASTER_CHANGELOG.md**, and **DB_SCHEMA.md** (or related schema documentation). Our focus is to remove redundancy, ensure consistency (especially after restructuring), and improve clarity. Below are the observations and recommendations for each:

- **README.md:** The README appears to be updated to reflect the current state of the application (for example, it notes the consolidation to a single app and new branding [28]). However, some sections may overlap with the changelogs or contain excess detail that can be streamlined.
- The README contains a "Project Overview" and installation instructions, which are great. We noticed it also includes a summary of architectural changes (retiring `concrete_mix_app`, etc.) that duplicate information from the CHANGELOG [28]. Consider removing or simplifying historical details in the README and instead focus it on *what the project is and how to use it now*. Historical changes can live in the changelog.
- Ensure any references to running scripts or managing data are updated given the new scripts organization. For example, if the README previously suggested running `run_server.py` to start the dev server, it should be updated to the standard `manage.py runserver` (since `run_server.py` will be archived). Similarly, if any data import instructions refer to scripts by name, update them to the new `scripts/` path or to using `manage.py import_ds`.

- Consistency: Check that terminology is consistent (the app is referred to as "CDB app" vs "Concrete Mix Database app" – pick one naming convention to avoid confusion). Also ensure the tone and detail level stay consistent (some parts read like release notes, which might be better kept in changelogs).

- **CHANGELOG.md:** This file logs software changes (we see version 1.0.0 and 1.0.1 entries) [29] [30]. It's focused on application changes and bug fixes. It appears to be in good order. One suggestion: ensure that any changes related to this branch's cleanup are added in a new changelog entry (e.g., "**Unreleased** – Database Refresh Cleanup: reorganized scripts into folders, no functional changes to app"). This helps transparency.

- Also, cross-check that entries in CHANGELOG.md don't conflict or duplicate MASTER_CHANGELOG entries. For instance, CHANGELOG 1.0.1 mentions consolidating to single 'cdb' database and removing obsolete templates [31], which MASTER_CHANGELOG also covers from a database perspective. This is acceptable as they serve different audiences (developers vs. DB admins), but ensure the facts are consistent (they are, as far as we can see).

- **MASTER_CHANGELOG.md:** This "master" changelog documents database operations, data fixes, and other behind-the-scenes changes [32] [33]. It's quite detailed (covering mix code standardization, material table cleanup, DS6 fixes, etc.). We find this separation (code changelog vs. data changelog) useful, as long as readers understand the distinction.

- To improve clarity, add a brief note at the top of MASTER_CHANGELOG.md explaining its purpose (e.g., "This changelog records data-level operations and migrations applied to the database, as a complement to the application CHANGELOG.md").
- Address redundancy: Some sections, especially about DS6, appear both here and in DS6_CHANGELOG.md (the dedicated DS6 log). For example, the material reference system implementation and purge process are described in both [34] [20]. It might be overkill to maintain DS6_CHANGELOG.md separately now that the information is consolidated in

MASTER_CHANGELOG. We recommend **merging DS6_CHANGELOG.md into MASTER_CHANGELOG.md** (as an appendix or detailed entry) or at least cross-referencing one from the other to avoid divergence. The DS6_CHANGELOG was useful during active troubleshooting, but going forward, a single source of truth for that story is easier to maintain.

- Ensure chronology and version tagging: The MASTER_CHANGELOG entries are timestamped. If the cleanup in this branch involves any data migrations (though it seems mostly code re-org), you might log a new entry like "May 2025 – Repository cleanup (no changes to data, organizational improvements to scripts)". This would note that thorough_ds6_purge.py and others were archived, etc., purely for tracking.

- **DB_SCHEMA.md:** This refers to documentation of the database schema. We did not find a file literally named DB_SCHEMA.md in the repo search, but we did find `cdb_database_dictionary.md` and `cdb_database_dictionary_generated.md`. These appear to serve the schema documentation purpose – the former is a manually curated data dictionary (as of Apr 23, 2025) [35], and the latter is an auto-generated schema Markdown (with a warning not to edit by hand) [36]. It's likely that DB_SCHEMA.md is intended to be one of these or to replace them.

- **Clarity:** Having two similar schema docs can confuse contributors. Decide on one primary schema reference. If the auto-generated one (`*_generated.md`) is comprehensive, consider linking to it from the manual one or vice versa, or merging them. For example, the manual `cdb_database_dictionary.md` could be renamed to **DB_SCHEMA.md** and contain high-level schema diagrams or explanations, while pointing readers to the auto-generated appendix for full field listings.
- **Redundancy:** Avoid maintaining the same content in two places. If the auto-generated doc is up-to-date (it includes model relationships and even record counts as of a certain date), perhaps that alone suffices. You could then drop the manually written table if it's not adding unique value. Alternatively, continue to use the manual one for descriptions while auto-generating the technical details periodically. Whichever path, make sure they don't contradict each other.

- **Consistency:** If there were schema changes during the database-refresh work (none major are evident except data fixes), ensure the documentation reflects those. For instance, if column names changed (the changelog mentions `test_value → value_num`, etc. [37]), the schema docs should reflect the new names consistently. Verify that DB_SCHEMA.md (or whichever file is canonical) lists the updated fields and notes any deprecations.

- **Other docs:** Although not explicitly asked, we glanced at **DS6_FIX_README.md** and **DS6_CHANGELOG.md**. They contain valuable narrative on the DS6 resolution. After integrating their essence into MASTER_CHANGELOG, these could be moved to an archive docs folder or left as-is for historical record. At minimum, ensure that any critical lessons or instructions from DS6_FIX_README.md (like the new material reference system steps) are captured in either the README (if user-facing) or a developer guide. Redundant or outdated instructions (especially the old fix instructions that were crossed out in DS6_FIX_README) can be clearly marked as obsolete.

Overall, the documentation is fairly thorough; our goal is to streamline it: keep each document focused and eliminate duplicate narratives. The README should tell a newcomer how to get started and what the project does. The CHANGELOG should list user-visible changes, and MASTER_CHANGELOG should log backend data operations. The schema docs should accurately describe the current database. With the repository reorganization, we will also update any documentation references to file paths (e.g., pointing out that utility scripts are now in `scripts/` subfolders).

Code Quality Improvement Opportunities

While reorganizing the scripts, we identified several code quality enhancements that can be applied across the board. Addressing these will make the scripts more maintainable and consistent:

- **Remove Unused Imports:** Many of these one-off scripts might carry over imports that are not needed (especially if they were copied from each other during frantic debugging). Performing an import audit on each script is advised. This cleanup makes the scripts clearer and avoids any side effects from unnecessary imports. For example, if `import pandas as pd` is in a script that no longer uses pandas, drop it.

- **Use Consistent Logging Instead of Prints:** It's likely that scripts like the purge or import utilities use simple `print()` statements for progress and output. Transitioning to the Python `logging` module would allow better control over log levels and integration with a logging config if needed. For instance, `thorough_ds6_purge.py` could log what it's deleting and any errors, rather than just printing to console – this would be crucial if run in a non-interactive environment. We suggest establishing a standard logging format (timestamped INFO/ERROR messages) for all scripts that perform significant actions. This consistency helps when reviewing outputs and debugging.

- **Add Argument Parsing (argparse):** Many scripts likely have hardcoded parameters (file paths, dataset identifiers, etc.) or require editing the script before each use. Introducing command-line arguments makes them much more flexible and safer to reuse. For example:

- `create_corrected_mapping.py` can accept arguments for input file and output file paths instead of requiring a code change for each dataset.
- `ds6_health_check.py` could take a dataset name or ID to generalize it beyond DS6, or a flag to output summary vs detailed results.

- If `manage_cdb.py` were to remain in use, it already parses `sys.argv` to pass through commands [38] [39] – we can ensure any new script with subcommands uses a similar approach or argparse subparsers.
  By using argparse in each script, we also automatically get `-h/--help` messages, which is user-friendly for future maintainers.

- **Refactor Duplicate Logic:** During the DS6 troubleshooting, multiple scripts may have implemented overlapping functionality (e.g., two different scripts scanning for mix codes, or linking materials in slightly different ways). Where feasible, consolidate this logic to avoid maintaining several variants. For instance, if both `check_mix_codes.py` and `check_database.py` were checking naming conventions, a single script (or even better, a Django management command or a function in the app) could provide a canonical implementation. Given that most of those duplicates are now archived, this is more of a forward-looking suggestion: in the future, try to extend or parameterize an existing script rather than create a new one from scratch for similar tasks.

- **Integrate with Django when Possible:** Some scripts (especially those doing database reads/writes) can leverage Django's ORM and model methods instead of raw SQL or manual psycopg2 connections. We noticed that the import process was integrated as a Django command (`import_ds.py` uses the Django environment to track materials, etc. [3]). Similarly, health checks and purges might benefit from being Django commands (so they run in the Django context with all models loaded). This would allow reuse of model logic (e.g., calling model `.objects.filter()` instead of writing raw queries) and ensure database transactions and settings are respected. As a code quality improvement, consider migrating critical scripts like `thorough_ds6_purge` into Django management commands in the future. This would give a more uniform interface (`manage.py thorough_ds6_purge`) and potentially safer execution (could use Django transactions, etc.).

- *However*, converting scripts to commands is a refactoring beyond simple cleanup, so this can be logged as a low-priority future enhancement. For now, the focus is on organizing what's there.

- **Consistent Coding Style and Comments:** Ensure all scripts have a short docstring at the top explaining their purpose, especially now that they'll be organized by folder (e.g., a script in `archived/` should mention it's deprecated, a script in `analysis/` should mention what analysis it performs). We saw some scripts like `run_server.py` and `manage_cdb.py` do include docstrings or printed usage instructions [40] [41] – that's good practice. We should standardize this: every script should at least have a comment or printout of how to use it. For example, adding an `if __name__ == '__main__': parser.print_usage()` for argparse-based scripts with no arguments can guide users.
  Also, follow PEP8 style consistently – e.g., proper naming, spacing, etc. (Most of these scripts are short-lived so style might have been overlooked under time pressure). A quick pass to fix indentation or long lines would polish them.

- **Testing and Verification:** If possible, test the key scripts after moving them to ensure they still run as expected. Some scripts might have file path assumptions (like expecting to be run from project root, using relative paths for data files). After moving, verify those paths. For instance, a script that opens `./etl/somefile.csv` will still find it if executed from root, but if one script was in `etl/` folder and assumed current directory accordingly, moving it might change that context. Adjust file paths in the scripts to be robust (perhaps constructing paths relative to a known base directory). This is a quality fix that prevents runtime errors post-reorg.

By implementing these code quality improvements, we not only clean the repository structure but also make each remaining script more robust and easier to use. This will reduce technical debt and ease onboarding of new contributors (they won't be scratching their heads over dozens of cryptically named Python files at the root).

Summary of Recommended File Actions

The table below summarizes the actions for each identified root-level script or relevant file, along with the justification and assessed risk level:

File (Script)	Proposed Action	Justification
`manage.py`	**Keep at root (no change)**	Critical Django entry point; moving would break app.

File (Script)	Proposed Action	Justification
`manage_cdb.py`	Move to `scripts/active/` (or archive)	Likely obsolete after single-DB consolidation; keep accessible just in case [10].
`run_server.py`	Move to `scripts/archived/`	Outdated dual-app dev script (original app retired) [11].
`create_corrected_mapping.py`	Move to `scripts/etl/`	Current ETL tool for mapping files (valuable for imports) [5].
`import_*.py` *(legacy import scripts, if any)*	**Archive**	Superseded by unified `import_ds.py` command [3].
`ds6_health_check.py`	Move to `scripts/validation/`	Data integrity check for DS6 (could be reused/adapted) [14].
`generate_ds6_health_flags.py`	Move to `scripts/analysis/`	Generates analysis metrics (data completeness) [15].
`check_mix_codes.py`	Move to `scripts/archived/`	Temporary DS6 diagnostic, no longer needed [1].
`find_all_mixes.py`	Move to `scripts/archived/`	Temporary DS6 diagnostic, no longer needed [1].
`check_database.py`	Move to `scripts/archived/`	Temporary DS6 diagnostic, no longer needed [1].
`fix_ds6_final.py`	Move to `scripts/archived/`	Obsolete DS6 fix attempt, replaced by new approach [1].
`link_material_properties.py`	Move to `scripts/archived/`	Obsolete DS6 fix attempt, replaced by new approach [21].
`validate_property_linking.py`	Move to `scripts/archived/`	Obsolete DS6 validation attempt [21].
`purge_ds6_data.py`	Move to `scripts/archived/`	Superseded by thorough_ds6_purge (partial solution) [21].
`thorough_ds6_purge.py`	Move to `scripts/database_management/` (**keep**)	Final DS6 purge script kept for reference [42].
`generate_schema_markdown.py` <br/>(if exists for schema doc)	Move to `scripts/analysis/` or `scripts/etl/`	Utility to generate DB schema documentation (matches `cdb_database_dictionary_generated.md`).
`create_column_map.py` (found in `etl/temp/`)	Relocate to `scripts/etl/`	Part of ETL process (likely used to create mapping templates).

File (Script)	Proposed Action	Justification
`test_*.py` *scripts (any)*	Move to `scripts/experimental/`	Not part of production; were for debugging or exploring.
SQL fix scripts (e.g. `fix_ds6_mixcodes.sql`*)*	Move to `scripts/database_management/` or `sql/`	Organizational – keep all DB fix scripts together for clarity [25].
Documentation (`README.md`, `CHANGELOG.md`, etc.)	**Update content** (not a move)	Remove redundancies, reflect new script paths, clarify schema docs.

Table: File-by-file cleanup actions for the database-refresh branch.

Legend: “Move” implies relocating the file into the specified `scripts/` subdirectory (creating it if necessary). “Archive” means moving to `scripts/archived/` and treating as deprecated (with no intention to run). Risk levels assume following our recommendations; “Low” means minimal chance of affecting application functionality, “Medium” indicates a need to double-check usage before change, “High” means do not modify (critical component).

Conclusion

By implementing the above recommendations, the *concrete-mix-database* repository will be far more organized and maintainable. The root directory will be uncluttered – containing only essential project files (Django settings, manage.py, etc.) and top-level folders – while all auxiliary scripts will be neatly categorized under `scripts/`. This structure makes it clear which scripts are actively used, which are for one-time data operations, and which are historical. Developers onboarding to the project can focus on the core application without distraction, yet still have access to the rich history of data fixes and tools in an orderly fashion.

In tandem, updating the documentation ensures that knowledge is centralized and easy to follow. Eliminating duplicate changelog content and clarifying the schema documentation will prevent confusion and reduce documentation upkeep effort.

Finally, the suggested code quality improvements (logging, argparse, etc.) will standardize how these scripts are written and used, reducing potential errors and making them easier to run and rerun. Although some of these scripts may never need to be executed again, bringing them up to a higher standard now means if and when they are revisited (or copied as a starting point for a new script), they will be easier to understand and trust.

Overall, these cleanup actions carry **low risk** to the application’s stability since they primarily affect ancillary files. By being conservative (archiving rather than deleting uncertain cases), we preserve all important work while still achieving the goal of a “spring-cleaned” codebase. The development team should experience immediate benefits in navigation and clarity, and the project will be in a better position for future development and dataset additions.

---

[1] [2] [3] [4] [5] [14] [15] [16] [17] [18] [19] [21] [22] [23] [24] [34] [42] DS6_CHANGELOG.md
https://github.com/kevinharperr/concrete-mix-database/blob/09c3167493b897be7470151ca59ffd7a06be3432/DS6_CHANGELOG.md

6 7 38 39 40 manage_cdb.py
https://github.com/kevinharperr/concrete-mix-database/blob/09c3167493b897be7470151ca59ffd7a06be3432/manage_cdb.py

8 9 10 12 13 29 30 31 37 CHANGELOG.md
https://github.com/kevinharperr/concrete-mix-database/blob/09c3167493b897be7470151ca59ffd7a06be3432/CHANGELOG.md

11 41 run_server.py
https://github.com/kevinharperr/concrete-mix-database/blob/09c3167493b897be7470151ca59ffd7a06be3432/run_server.py

20 25 26 32 33 MASTER_CHANGELOG.md
https://github.com/kevinharperr/concrete-mix-database/blob/09c3167493b897be7470151ca59ffd7a06be3432/MASTER_CHANGELOG.md

27 ols_baseline.txt
https://github.com/kevinharperr/concrete-mix-database/blob/09c3167493b897be7470151ca59ffd7a06be3432/analysis_output/ols_baseline.txt

28 README.md
https://github.com/kevinharperr/concrete-mix-database/blob/09c3167493b897be7470151ca59ffd7a06be3432/README.md

35 cdb_database_dictionary.md
https://github.com/kevinharperr/concrete-mix-database/blob/09c3167493b897be7470151ca59ffd7a06be3432/cdb_database_dictionary.md

36 cdb_database_dictionary_generated.md
https://github.com/kevinharperr/concrete-mix-database/blob/09c3167493b897be7470151ca59ffd7a06be3432/cdb_database_dictionary_generated.md