County of Santa Cruz

HEALTH SERVICES AGENCY
Environmental Health Division
701 Ocean St. Room 312, Santa Cruz, CA 95060
(831) 454-2022 TDD/ TTY: Call 711
EnvironmentalHealth@santacruzcountyca.gov
www.scceh.org

AI-OCR Well Layer Update Project

Executive Summary:

Project Overview

The AI-OCR Well Layer Update Project was undertaken to modernize the County of Santa Cruz's GIS well layer—a foundational dataset for groundwater and water-resource management. Funded through the 2023 County Drought Resilience Planning Grant, the project applied AI-powered optical character recognition (OCR), custom Python scripts, GIS workflows, and structured quality-assurance (QA) methods to extract, clean, and integrate data from over 17,000 historical well completion files (representing approximately 10,000 unique wells). Many of these reports were decades-old, scanned PDFs with handwritten or inconsistent entries. The project's automated workflows brought them into a standardized, GIS-ready format to support more accurate analysis, planning, and long-term resilience.

Objectives

- Improve completeness and accuracy of historical well records.
- Standardize key fields including well ID, well type, depth, water levels, locations, and screen intervals.
- Align County, OCR derived, and Department of Water Resources (DWR) datasets for GIS integration.
- Reduce manual data-entry workload and significantly improve geocoding accuracy.

Approach

The project combined a custom-trained Microsoft Azure Document Intelligence model, six purpose-built Python scripts, fuzzy matching and aliasing logic, and multi-layered QA workflows to digitize, clean, classify, and integrate over 17,000 scanned well completion reports. ArcGIS-based workflows then geocoded wells to the highest achievable accuracy, with confidence flags for transparency.

Key Achievements

- **Records Processed:** 17,672 well completion reports (7,046 County-held; 10,626 DWR) processed in under one day—saving thousands of hours of manual effort.
- **Unique Wells Identified:** Created the most complete countywide inventory to date: 10,007 unique, deduplicated wells, including 713 previously unrecorded wells discovered in County archives.
- **Dataset Alignment:** Reconstructed 95% of DWR entries directly from scanned originals.
- **Attribute Completeness:**
    - OCR workflows filled 78% of missing drill depths.
    - Improved extraction of static water level, depths, yield, screen intervals, drilling methods, and locations.
    - Classified wells into production (6,077), destruction (2,185), monitoring, and additional types.
- **Location Geocoding Improvements:**
    - 91% of wells geocoded beyond PLSS centroid placement.
    - 5,799 used precise County-verified coordinates;
    - 3,288 mapped to parcel centroids;
    - 898 mapped using PLSS or manual fallback.

    - New “Exact” vs. “Approximate” vs “PLSS” confidence flags improve spatial transparency.
- **Duplicate Resolution:** Resolved distinct 66 duplicate IDs and more than 6,000 redundant scans, improving dataset quality.

Quality Assurance (QA) Highlights

- Approximately 15% of AI supplemented records were flagged for review based on issues such as:
    - Conflicting depth measurements
    - Well type inconsistencies
    - Low-confidence APN/address matches
- Red-highlighted AI-derived fields, confidence scoring, and structured review queues ensured efficient and reliable manual verification.

Outcomes & Impact

The project produced the most accurate, complete, and spatially reliable well layer ever assembled for Santa Cruz County. The dataset, which is published to the County’s public portal, GISWeb, now supports:

- Drought resilience planning
- Groundwater monitoring and basin assessment
- Well permitting and oversight
- Long-term water resource protection

The workflow is reproducible, scalable, and statewide-ready, offering a model for other counties and agencies seeking to modernize archival water datasets.

Conclusion

By integrating AI-OCR, Python automation, QA, and GIS, the County transformed decades of handwritten, inconsistent, and poorly geocoded records into a unified, high-confidence dataset. The project dramatically reduced manual workload, increased spatial and attribute completeness, and demonstrated how modern AI tools can convert legacy water records into actionable information for sustainable groundwater management.

The remainder of this report expands on these findings, outlining the methods, data processing steps, QA approach, and final outcomes in greater detail.

Contents

Introduction

The AI-OCR Well Layer Update Project modernized the County of Santa Cruz's GIS well layer, a critical tool for water resource management within the County, by automating data extraction and matching from well completion reports. Funded through the 2023 County Drought Resilience Planning Grant, the project combines AI-powered OCR, custom Python scripting, QA procedures, and GIS integration to significantly improve data accuracy and completeness. Initiated in 2024, this report outlines the project's methodology, including model training, six core data processing scripts, QA workflows, and GIS updates, as well as the resulting improvements.

Background and Challenge

Since its creation in 2005, through the merger of County and partner agency datasets, the County of Santa Cruz's GIS well layer has faced persistent issues with incomplete and inconsistent information. A 2009 integration with the DWR well layer provided some improvements; however, more than 75% of the 11,700 wells still lacked well type classifications, and a significant number of records were missing altogether. In addition, several wells were entered into the dataset more than once. These gaps and inconsistencies have limited the Water Resources Division and other agencies' ability to accurately identify production wells and assess their impacts.

The current project was designed to address these data gaps and more, by applying AI-powered OCR to scanned well completion reports. Using custom scripts, the project extracted, cleaned, and standardized key fields—such as well ID, depth, water levels, APN, screen intervals, and address—and employed fuzzy matching to align these records with the existing DWR's well database. This process enhanced data completeness and consistency while preserving the County's existing spatial accuracy and retaining the county-specific fields

we still find valuable, such as whether an electric log is associated with the well record.

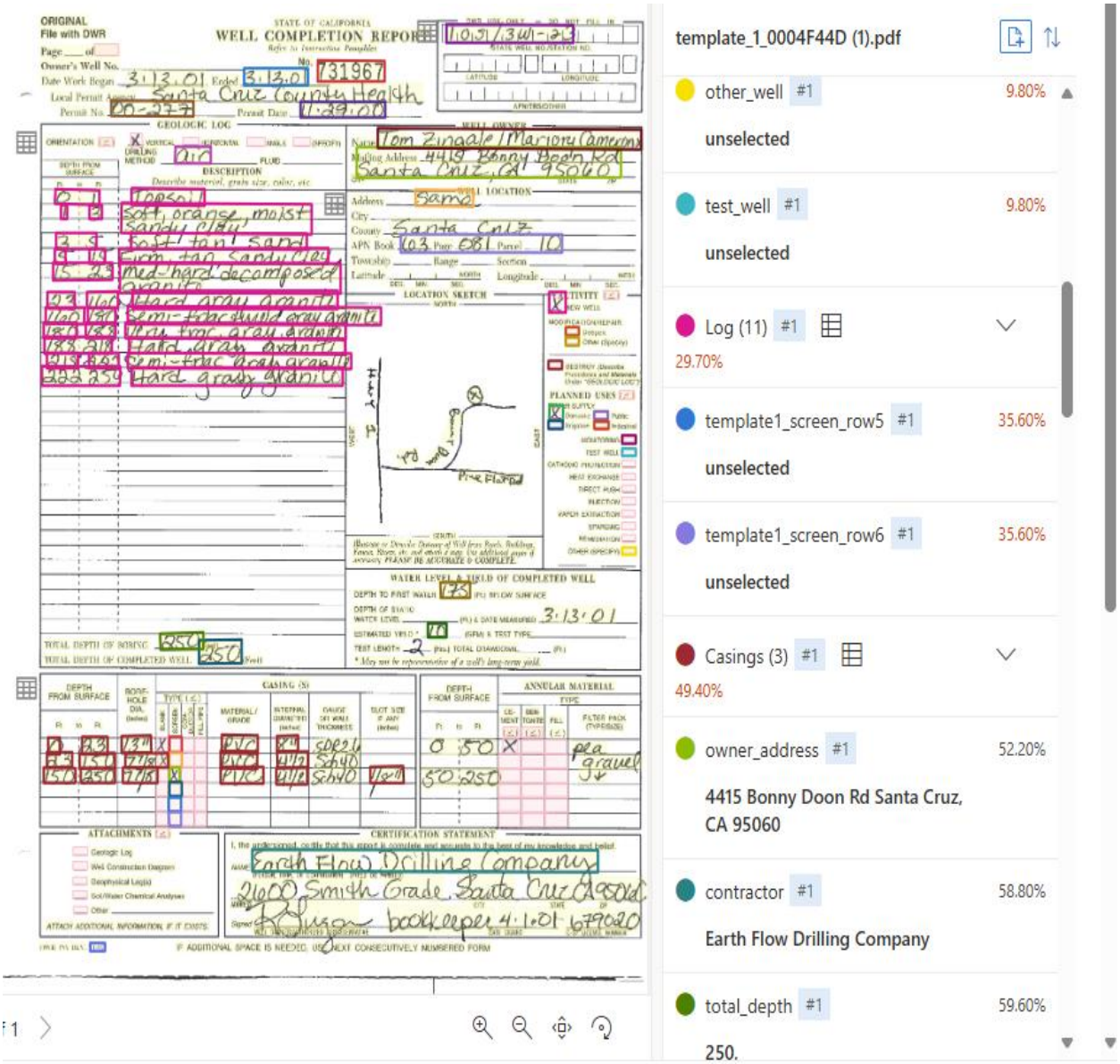
Methodology

The project employed Microsoft Azure Document Intelligence for OCR, supported by six custom Python scripts for data processing, continuous QA to validate outputs, and GIS integration to update and visualize the results. The overall workflow is summarized below, with a visual overview of the script sequence provided in the attached diagram (*Script Sequence Workflow,* Appendix Page 1).

Model Training

A custom Azure Document Intelligence model was trained using nine standardized DWR well completion report templates, dating back to the late 1940s to early 1950s. The model was designed to extract key fields such as well ID, total depth, water levels, completion dates, yields, well type, seals, lithology, screen intervals, APNs and other locational information, and more. It was optimized to handle a wide range of report formats—including handwritten entries and low-resolution scans—ensuring consistent and accurate data extraction across standard California DWR reports.

An example of the model's digitized output, displaying color-coded bounding boxes for extracted data and a right-hand panel listing some of the parsed fields, is provided in the following Azure Document Intelligence screenshot:



Data Processing Scripts

Six custom Python scripts were developed to process, clean, classify, and enhance the data extracted through OCR. The scripts leverage AI-assisted coding tools (e.g., ChatGPT, Copilot) to accelerate prototyping and debugging. They utilize libraries including pandas, fitz, tempfile, numpy, os, json, logging, re, collections, rapidfuzz, azure-ai-formrecognizer, and openpyxl.

The scripts work sequentially to extract structured data from scanned well reports, clean and standardize fields, classify well types, match records

across multiple datasets, integrate county records, and highlight AI-derived fields for quality control. The workflow for each script is summarized below.

1. OCR Extraction Script

Purpose: Convert raw PDF well completion reports into structured JSON data suitable for downstream processing.

Overview:
Processes scanned PDF well reports using Azure Document Intelligence for OCR, with fitz handling PDF manipulation and compression.

Key Functions:

- Compresses large PDFs (>4 MB) to ensure compatibility with Azure OCR.
- Extracts fields such as well_id, depth, APN, and Casings using the custom-trained Azure model.
- Captures page- and line-level content in nested JSON dictionaries.
- Logs extraction errors and skips failed PDFs.
- Outputs JSON files to specified directory.

2. Data Extraction and Classification Script

Purpose: Transform JSON outputs from OCR into structured Excel tables, with keyword detection to support well type classification.

Overview:
Processes JSON outputs from the OCR Extraction Script, extracting key fields and identifying destruction-related keywords. Outputs two Excel sheets: *Main Data* and *Casings*.

Key Functions:

- Parses JSON dictionaries, extracting fields like well_id, destruction_well, address, and APN.
- Detects destruction-related keywords (e.g., “abandonment,” “destroy,” “decommission”) using fuzzywuzzy (token_sort_ratio threshold 85), including common misspellings.

- Converts Casings lists into DataFrame rows with traceability via source_file and row_index.
- Skips invalid JSON structures and logs errors.
- Generates Excel file output to specified directory.

3. Data Cleaning and Matching Script

Purpose: Produce a cleaned, deduplicated dataset with standardized fields, enhanced classifications, and matched geospatial data.

Overview:
Processes the Excel output from extraction to clean, standardize, and enrich data. Handles OCR errors, refines well classifications, extracts perforation intervals, and prepares addresses/APNs for matching.

Key Functions:

- Cleans well_id using regex, handling OCR misreads.
- Classifes templates (1–9 or Unknown) via revision-based signatures.
- Refines destruction classifications using baseline keyword lists.
- Extracts perforation intervals and validates top/bottom values.
- Validates the values of Total Completed Depth and Total Drill Depth, ensuring that Total Completed Depth is always greater than Total Drill Depth.
- Detects duplicates using weighted scoring and retains the most complete record.
- Cleans addresses and other fields (e.g. City, Driller Name, Fluid Type, etc.) via dictionary-based aliases and regex;
- Searches for and extracts APNs from the document (which may be located in multiple places within the document).
- Derives Type of Work hierarchically from fields like keyword_destruction_well or new_well.
- Handles invalid numeric fields (NaN) and missing data.
- Generates Excel file output to specified directory.

4. Data Matching and Merging Script

Purpose: Align OCR-derived well records with the DWR database, enhance APN and address accuracy, and produce a comprehensive matched dataset.

Overview:
Integrates cleaned OCR data with DWR records and optionally checks Monterey County or other County adjacent datasets, generating multiple output files for matched and unmatched records.

Key Functions:

- Cleans well IDs and normalizes APNs (handling 8- and 9-digit formats).
- Performs primary matching on well IDs, validated by depth or water level.
- Multiple matches are resolved with a weighted scoring algorithm
- Refines APNs and addresses using parcel validation and fuzzy matching techniques.
- Validates DWR APNs against a dictionary mapping to parcel addresses. If an APN is invalid, it extracts APNs from the Well Location or other fields if present, or employs rapidfuzz for address matching. Final APNs are formatted to align with the parcel database standards used for GIS integration.
- Checks unmatched OCR records against adjacent County datasets.
- Outputs:
    - unmatched_ocr_output.xlsx
    - unmatched_dwr_output.xlsx
    - matched_output.xlsx
- Maintains source tracking and logs matching details for QA.

5. County Database Matching Scripts

Purpose: Integrate county database records with DWR or unmatched OCR data, validating and enriching well records.

Overview:
Two scripts align county records with either DWR data or OCR data, producing matched and unmatched outputs.

Key Functions:

- Normalizes county and source IDs using regex (handling OCR misreads and variations).
- Matches county Log_Number to DWR Legacy Log Number/WCR Number or OCR well_id_cleaned.
- Prevents duplicate matches with sets and logs metadata.
- Produces three Excel outputs per script: matched records, unmatched county records, and unmatched source records.

6. Field Highlighting Script

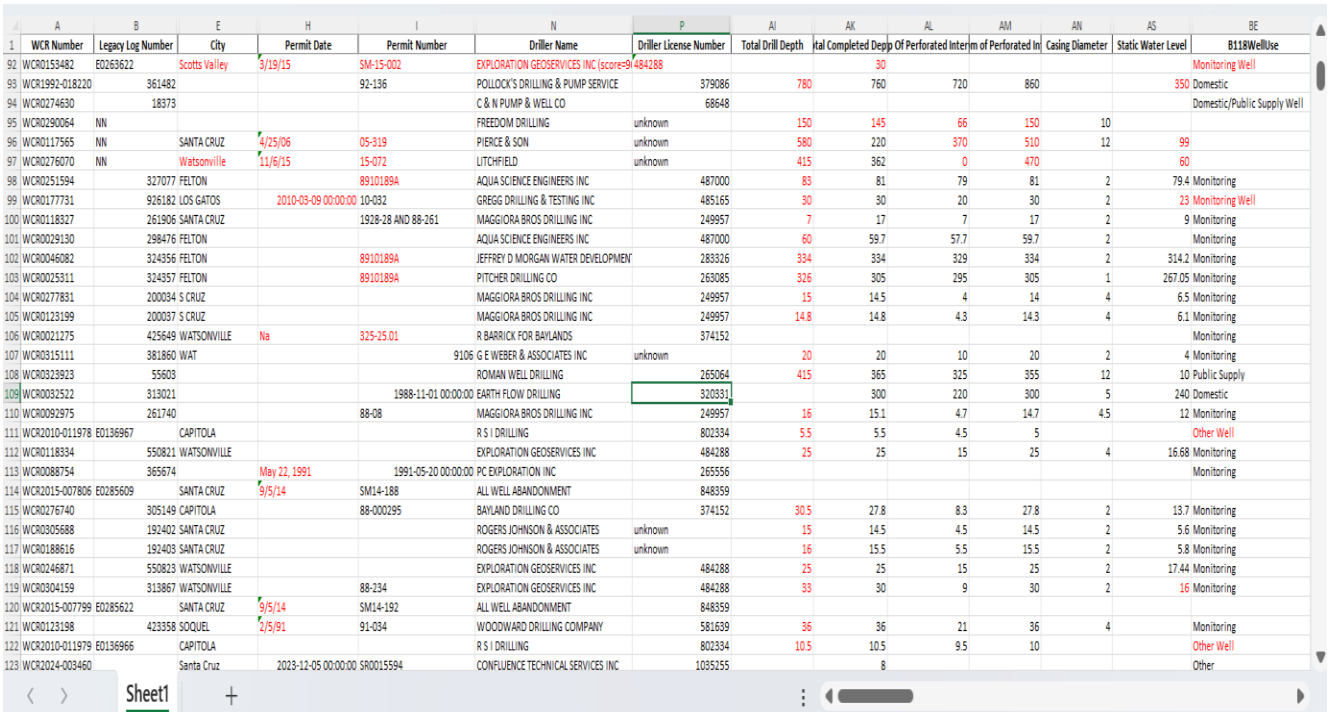
Purpose: Visually highlight OCR-derived or non-validated fields in the final supplemented dataset for QA based on the data source.

Overview:
Applies conditional formatting to the supplemented dataset Excel file using openpyxl, marking AI-sourced fields or unmatched values in red font.

Key Functions:

- Loads Excel workbook and maps columns dynamically.
- Highlights final_apn and final_address if match_source indicates non-validated or OCR-derived data.
- Highlights other fields based on a predefined field_map when source = "AI."
- Skips missing columns and logs errors.
- Saves the modified Excel file, overwriting the input file.

The following image shows an example output of the field-highlighting script, with color-coded AI-generated values:



	A	B	E	H	I	N	P	AI	AK	AL	AM	AN	AS	BE
1	WCR Number	Legacy Log Number	City	Permit Date	Permit Number	Driller Name	Driller License Number	Total Drill Depth	tal Completed Dep	p Of Perforated Inter	m of Perforated In	Casing Diameter	Static Water Level	B118WellUse
92	WCR0153482	E0263622	Scotts Valley	3/19/15	SM-15-002	EXPLORATION GEOSERVICES INC (score=9	484288		30					Monitoring Well
93	WCR1992-018220	361482			92-136	POLLOCK'S DRILLING & PUMP SERVICE	379086	780	760	720	860		350	Domestic
94	WCR0274630	18373				C & N PUMP & WELL CO	68648							Domestic/Public Supply Well
95	WCR0290064	NN				FREEDOM DRILLING	unknown	150	145	66	150	10		
96	WCR0117565	NN	SANTA CRUZ	4/25/06	05-319	PIERCE & SON	unknown	580	220	370	510	12	99	
97	WCR0276070	NN	Watsonville	11/6/15	15-072	LITCHFIELD	unknown	415	362	0	470		60	
98	WCR0251594	327077	FELTON		8910189A	AQUA SCIENCE ENGINEERS INC	487000	83	81	79	81	2	79.4	Monitoring
99	WCR0177731	926182	LOS GATOS	2010-03-09 00:00:00	10-032	GREGG DRILLING & TESTING INC	485165	30	30	20	30	2	23	Monitoring Well
100	WCR0118327	261906	SANTA CRUZ		1928-28 AND 88-261	MAGGIORA BROS DRILLING INC	249957	7	17	7	17	2	9	Monitoring
101	WCR0029130	298476	FELTON			AQUA SCIENCE ENGINEERS INC	487000	60	59.7	57.7	59.7	2		Monitoring
102	WCR0046082	324356	FELTON		8910189A	JEFFREY D MORGAN WATER DEVELOPMENT	283326	334	334	329	334	2	314.2	Monitoring
103	WCR0025311	324357	FELTON		8910189A	PITCHER DRILLING CO	263085	326	305	295	305	1	267.05	Monitoring
104	WCR0277831	200034	S CRUZ			MAGGIORA BROS DRILLING INC	249957	15	14.5	4	14	4	6.5	Monitoring
105	WCR0123199	200037	S CRUZ			MAGGIORA BROS DRILLING INC	249957	14.8	14.8	4.3	14.3	4	6.1	Monitoring
106	WCR0021275	425649	WATSONVILLE	Na	325-25.01	R BARRICK FOR BAYLANDS	374152							Monitoring
107	WCR0315111	381860	WAT		9106	G E WEBER & ASSOCIATES INC	unknown	20	20	10	20	2	4	Monitoring
108	WCR0323923	55603				ROMAN WELL DRILLING	265064	415	365	325	355	12	10	Public Supply
109	WCR0032522	313021			1988-11-01 00:00:00	EARTH FLOW DRILLING	320351		300	220	300	5	240	Domestic
110	WCR0092975	261740			88-08	MAGGIORA BROS DRILLING INC	249957	16	15.1	4.7	14.7	4.5	12	Monitoring
111	WCR2010-011978	E0136967	CAPITOLA			R S I DRILLING	802334	5.5	5.5	4.5	5			Other Well
112	WCR0118334	550821	WATSONVILLE			EXPLORATION GEOSERVICES INC	484288	25	25	15	25	4	16.68	Monitoring
113	WCR0088754	365674		May 22, 1991	1991-05-20 00:00:00	PC EXPLORATION INC	265556							Monitoring
114	WCR2015-007806	E0285609	SANTA CRUZ	9/5/14	SM14-188	ALL WELL ABANDONMENT	848359							
115	WCR0276740	305149	CAPITOLA		88-000295	BAYLAND DRILLING CO	374152	30.5	27.8	8.3	27.8	2	13.7	Monitoring
116	WCR0305688	192402	SANTA CRUZ			ROGERS JOHNSON & ASSOCIATES	unknown	15	14.5	4.5	14.5	2	5.6	Monitoring
117	WCR0188616	192403	SANTA CRUZ			ROGERS JOHNSON & ASSOCIATES	unknown	16	15.5	5.5	15.5	2	5.8	Monitoring
118	WCR0246871	550823	WATSONVILLE			EXPLORATION GEOSERVICES INC	484288	25	25	15	25	2	17.44	Monitoring
119	WCR0304159	313867	WATSONVILLE		88-234	EXPLORATION GEOSERVICES INC	484288	33	30	9	30	2	16	Monitoring
120	WCR2015-007799	E0285622	SANTA CRUZ	9/5/14	SM14-192	ALL WELL ABANDONMENT	848359							
121	WCR0123198	423358	SOQUEL	2/5/91	91-034	WOODWARD DRILLING COMPANY	581639	36	36	21	36	4		Monitoring
122	WCR2010-011979	E0136966	CAPITOLA			R S I DRILLING	802334	10.5	10.5	9.5	10			Other Well
123	WCR2024-003460		Santa Cruz	2023-12-05 00:00:00	SR0015594	CONFLUENCE TECHNICAL SERVICES INC	1035255		8					Other

Sheet1

Note: See the Appendix for the code and detailed function descriptions.

Quality Assurance

The Well Layer Update data processing workflow incorporates multiple layered quality assurance mechanisms to improve accuracy, reliability, and traceability. These mechanisms operate at the field-cleaning, matching, and validation stages and generate flags, scores, source tracking, and visual cues that guide manual review. Outputs from the Data Cleaning and Matching Script, Data Matching and Merging Script, County Database Matching Scripts, and Field Highlighting Script include dedicated QA columns, confidence metrics, and conditional formatting designed to highlight low-confidence or potentially erroneous values.

The key QA mechanisms are summarized below.

1. **Location Matching Confidence and Source Tracking**

Purpose: Provide transparent, quantifiable confidence in the final APN and address used for geocoding, enabling targeted manual review of lower-quality locations.

Implementation:

- Matching follows a strict hierarchy: exact validated APN → exact validated address → fuzzy validated APN → fuzzy validated address → fallback methods.
- Every record receives:

  – match_source / OCR_match_source: a human-readable string describing the match method and score (e.g., "ID match → APN validated", "Fuzzy address match (score 78.2%)")

  – Numeric fuzzy scores (0–100) from rapidfuzz when fuzzy address or APN parcel is used.

  – Alias/regex confidence scores when street-name or city dictionaries are applied during address cleaning.

QA Role: Records with fuzzy scores <85, non-APN-based matches, or low alias/regex scores (<80%) are prioritized for manual review using GIS tools and source PDFs.

2. **AI Source Indicators for OCR-Derived Data**

Purpose: Distinguish OCR-derived data from DWR or county-validated data.

Implementation: Scripts track OCR-derived fields using source columns (e.g., apn_source, depth_source) marked "AI." The Field Highlighting Script applies red font to these cells (e.g., final_apn, final_address, seal_cleaned, template_id).

QA Role: Red highlights visually flags potentially error-prone data, prompting user to verify against the original PDFs.

3. **Type Mismatches for Record Consistency**

   Purpose: Detect discrepancies between DWR and OCR-derived well type classifications.

   Implementation: The Record_Type_Mismatch field flags conflicts between DWR B118WellUse and OCR-derived types.

   QA Role: Prioritize mismatched records and cross-reference source documents to resolve classification errors.

4. **Destruction Well Cross-Referencing**

   Purpose: Improve destruction well accuracy, especially when handwritten notes override checked boxes (common when a proposed well was abandoned after drilling began).

   Implementation: keyword_destruction_well refines the initial checkbox detection by comparing additional keyword counts against template-specific baselines.

   QA Role: Records where OCR and keyword indicators disagree and are flagged for manual confirmation.

5. **Total Depth Validation Flag (depth_flag / review_depths)**

   Purpose: Ensure consistency between Total Drill Depth and Total Completed Depth and catch common OCR or data-entry errors.

   Implementation:

   - Validates that Total Drill Depth ≥ Total Completed Depth (with minor tolerance).
   - Sets depth_flag = 'review depths' when the relationship is violated or values appear swapped.

6. **Dictionary-Based Alias Cleaning with Regex Confidence Scoring**

   Purpose: Standardize inconsistent or misspelled categorical/text fields (City, Driller Name, Fluid, Test Type, Drilling Method, etc.) while quantifying confidence.

   Implementation:

   - Large alias dictionaries map historical and variant names to standardized values.
   - Two-stage cleaning: (1) dictionary lookup with rapidfuzz token-set ratio, (2) regex pattern fallback if no strong dictionary hit.
   - Each cleaned value receives an alias_match_score (0–100)

Manual Review Process

The QA outputs feeds into a structured review workflow:

- Red-highlighted AI-sourced fields (Field Highlighting Script)
- depth_flag = True or conflicting depth measurements
- Low dictionary/regex alias match scores (<80%) on City, Driller Name, Fluid Type, Drilling Method, etc.
- Record_Type_Mismatch or conflicting destruction indicators
- Fuzzy address/APN match scores <85 (especially non-APN or direct address matches)
- OCR_match_source indicating fallback or low-confidence location methods
- B118WellUse anomalies possibly caused by 'Confidential' stamp interferences

Verification Tools: ArcGIS Pro, original scanned PDFs, county parcel records and historical lists are utilized to update flagged record values.

GIS Workflow

The GIS workflow for the Well Layer Update emphasizes maximizing spatial accuracy of well locations by integrating updated database outputs with county-maintained spatial layers. It combines automated geocoding, join operations, and QA field value comparison checks to produce a high-confidence, GIS-ready dataset. The workflow prioritizes county-verified locations, leverages parcel-based centroids for unmatched (county) APN records, and applies PLSS grid centroids or manual location sketch plotting as fallbacks for records with missing or low-confidence location data.

Geocoding Process

1. Matching with County-Mapped Well Records:

Purpose: Preserve the accuracy of well locations already verified and mapped by county staff.

Implementation:

- Updated database records that match existing county-mapped wells are spatially joined based on Log_Number and well_id_cleaned.
- The county well layer geometry, established over years from DWR report records and maps, is preserved.
- County unique fields such as county_ELOG are appended to matched records.

QA Role:

- High-confidence matches inherit county-verified locations, reducing the need for further location verification.

2. APN and Address-Based Geocoding with Parcel Layer

Purpose: Assign the most precise possible locations to wells using validated APNs or addresses, while distinguishing exact from approximate placements.

Implementation:

- Records with a validated APN or address (cross-referenced against the County's Parcel database) are spatially joined to the centroid of the corresponding parcel polygon.
- Each record receives a Location_Accuracy field with one of two values:
    - "Exact" – assigned when the match is based on an exact, validated APN or address match, or inheritance from an existing county-mapped well.
    - "Approximate" – assigned when the match relies on fuzzy APN or address matching with a fuzz score <100 but ≥80–85

3. Fallback to PLSS Grid or Manual Review

Purpose: Provide approximate locations for records that are lacking or have low confidence APN/address data.

Implementation:

- PLSS Grid: Wells are mapped to the centroid of their corresponding PLSS Section grid in the county.
- Manual Review with Location Sketches: Extract well positions (ongoing) from sketches in DWR reports into ArcGIS.
- Records with low (<80) OCR_match_source fuzzy scores (e.g., "Address matched (score 63.64)") are flagged for fallback or manual review processing before mapping to that address or PLSS.

QA Role:

- Wells without confident location data are mapped via PLSS centroids and are subject to future cross-checks with the source file to improve location data manually.

GIS Summary:

This GIS workflow ensures well locations in the Well Layer Update are geocoded with maximum accuracy by:

1. Prioritizing county-verified coordinates.
2. Leveraging parcel-based centroids for direct match or high-confidence APN records.
3. Applying existing DWR PLSS grid mapping for low-confidence or missing data.

Results and Discussion

The project processed 17,672 well completion reports (7,046 county, 10,626 DWR) in batches using the developed Python scripts, achieving improvements in data integration, accuracy, and efficiency. The following results and metrics highlight improvements across data processing, matching, geocoding, field supplementation, QA, and GIS integration.

Data Processing

- Total Records Processed: Successfully extracted data from 17,672 files, with nearly all files processed without errors, despite challenges with low-quality scans and illegible handwritten data. Note some of these files ended up not being well completion reports (i.e. e-logs, site plans, boring logs, etc.)

Matching Accuracy

- County-to-DWR Matching: 5,683 of the 6,698 county-scanned well completion reports (85%) were successfully matched to corresponding DWR records using a combination of exact well ID matching and fuzzy reconciliation techniques.
- The remaining ~15% (1,015 records) were unmatched due to either the record not existing in the DWR database or poor scan quality or illegible handwriting that prevented accurate well ID extraction.
- As of April 28, 2025, the DWR online system for WCR contained 9,406 well records geographically associated with Santa Cruz County. Around the same date, 10,651 PDF files were downloaded from the DWR website for processing.
- After the initial county-to-DWR matching, 3,723 DWR records remained unsupplemented by county scans (9,406 total DWR records – 5,683 county-matched). Of these, 3,070 were subsequently matched and enriched using OCR data extracted directly from the downloaded DWR-sourced PDFs (following deduplication in the Data Cleaning and Matching Script). This left only 653 DWR records without OCR-derived supplemental data.
- Unique and Non-Well Records: 713 county-scanned records had no corresponding entry in the DWR database (true new records).
- ~530 county files were identified during QA review as non-well completion reports (e.g., boring logs, site plans, or electric logs) or were illegible and excluded from final counts.
- The OCR extraction and processing pipeline, when applied to the 10,651 PDFs downloaded from DWR's OSWCR website, successfully recreated 95% of the 9,406 official DWR records for Santa Cruz County using only the digitized documents themselves. When combined with the 713 previously unrecorded wells identified from county-held scans, the project achieved 103% relative completeness compared to the official Santa Cruz County DWR database as of April 2025.
- The updated master database contains 10,007 unique well records. This final dataset is a synthesis of DWR records enriched with AI-OCR-

extracted data, 713 previously unrecorded wells identified from county-held scans, and county records that were supplemented and validated using OCR.

Location Accuracy and Improvements

- Geocoding Success: Of the 10,007 wells in the updated dataset, 9,135 (91%) now have geocode locations ranging from exact well locations to approximate well locations—a significant improvement over the previous layer.
    - County-Mapped Wells: 5,799 matched records inherited precise county coordinates through GIS joins.
    - APN-Based Geocoding: 3,288 wells were mapped to parcel centroids, with 80% (2,644) validated as high-confidence (e.g. "APN matched") and the remaining 644 as approximate locations.
    - PLSS/Manual Fallback: 898 wells with low fuzzy-match scores (<80%) or missing location data were mapped to PLSS centroids, flagged for later QA review.
    - The remaining 48 inherited locations from DWR associated with GPS, Other, or Surveyed to Benchmark.

Well Type Improvements

The previous County well database had limited and inconsistently defined well type information. It often wasn't clear whether a well was a production well—or what type of production it served (domestic, irrigation, industrial, or municipal)—and it also did not reliably distinguish destruction wells from 'other' categories such as test, monitoring, or boring wells.

The updated database now classifies each well type based on the information available in the well completion reports, resulting in a more accurate and consistent countywide dataset. The updated counts include 6,077 production wells, 2,185 destruction wells, and 1,747 wells classified as other types.

Field Supplementation

- Critical Fields Filled: The Data Matching and Merging Script supplemented previously blank or unknown DWR fields using OCR and county data:

Field	Records Filled	Total Matched	Percent Filled
Well completion date	513	8,752	5.9%
Total drill depth	6,852	8,754	78.3%
Final well depth	405	8,754	4.6%
Static water level	545	8,754	6.2%
Well yield	1,074	8,754	12.3%
Well use/source	1,788	8,754	20.4%
Drilling method	896	8,754	10.2%
Screen intervals	670	8,754	7.7%

- Unique Fields: Additional fields (e.g., well seals, first water levels encountered, geologic materials) were extracted. Note: geologic materials were not processed or incorporated in any of the output datasets.

Duplicate Cleanup

- Duplicate Resolution: The Data Cleaning and Matching Script reconciled 66 distinct duplicate well IDs—cases where the same well log ID referred to different wells—and eliminated more than 6,000 duplicate PDF scans, significantly reducing redundancy in the dataset.

Time Savings

- Processing Efficiency: 17,672 reports were processed in under one day, saving significant amounts of time compared to manual extraction and geocoding, equivalent to thousands of hours saved if that would have been required to manually supplement or input the data.
- Script Runtime Breakdown:
    - OCR Extraction Script: ~70% of runtime
    - Post-Processing Scripts: ~30% hours, optimized by batch processing and indexing

Quality Assurance Metrics

QA-Flagged Records: Approximately 15% of all records (2,651 of 17,672) were flagged for manual review. Key flag categories included:

- ~2000 records affected by *"Confidential"* stamp interference (misread numbers, checkbox errors).
- 393 records with conflicting depth measurements (review_depths flag).
- 181 records with well type mismatches (often caused by *"Confidential"* stamp interference).

- 1,857 records with fuzzy match scores below 80% (from OCR_match_source).
- Roughly 5% of supplemented fields using the street-name alias dictionary generated low alias match scores (<80%), requiring manual refinement.

Challenges and Solutions:

- Poor scan quality / illegible handwriting affected numerous reports, mitigated through data cleaning, OCR error logging, and QA.
- Template inconsistencies: Handled by template-specific cleaning logic
- OCR errors: Minimized using a combination of fuzzy matching, dictionary-based aliasing, standardized or pattern-based cleaning routines, and QA-flagged review.

Scalability

- Statewide-Ready Framework**:**

  The California DWR template-based processing, APN integration, and GIS workflows create a scalable framework that can be extended to other counties or applied to statewide DWR datasets.

Potential Enhancements

Model Training & Template Improvements

- Train additional documents to improve value extraction, especially for Template 2.
- Add training for all selection boxes across templates (equipment-type checkboxes, cathodic protection wells, injection wells, etc.) to prevent lesser-used boxes from interfering (common) with more critical selections located nearby.
- Additional training across all known templates to address missed or unrecognized templates.
- Retrain the AI-OCR model to better detect monitoring wells and other specialized well types.
- Refinement of template classifications.
- Improved template-specific logic for screen interval extraction, especially for Template 1.
- Template-specific well ID cleaning, such as enforcing expected ID ranges (e.g., Template 2).

OCR & Document Handling

- Improved OCR performance for low-quality or complex scans.
- Automated detection and removal of “Confidential” stamps.
- Enhanced parsing and handling of PDFs containing multiple well completion reports.
- Add logic to detect and classify documents that are not well completion reports (borings, e-logs, site plans, etc.).

Data Cleaning & Field Logic Enhancements

- Strengthen cleaning algorithms for Total Depth, including:
    - cross-checking Total Drill Depth against the bottom depth in the geologic log,
    - handling “Same” entries,
    - applying fallback logic to populate final depth from geologic log values when needed.
- Add logic and fields for detailed screen interval depths.
- Add lithology post-processing.
- Enhance screen interval logic (e.g., ensure the screen checkbox aligns with slot-size entries in the same row).
- Add a new field to classify wells that were attempted but ultimately dry (Files that contain:“Dry”, “Dry Hole”, “Dry Well”).
- Add a new field to identify wells with accompanying electric logs (e.g., files containing terms such as “Electric Log,” “E-Log,” or “Elog,” geophysical logs marked as checked, or forms where “Was electric log made?” is checked).
- Logic to classify wells with no “Proposed Use” selected as Production if a screen interval exists, with a QA flag to ensure they aren’t monitoring wells.
- More robust handling of duplicate well IDs.

Quality Assurance Enhancements

- Increased QA checks comparing AI-extracted data with DWR records to identify and correct errors

- Add built in QA checks (rather than relying later on GIS checks which are more time intensive) to flag impossible or inconsistent values, such as:
    - static water level or first water deeper than total drill depth,
    - bottom of perforated interval deeper than total drill depth.
- Several DWR data entry errors were identified during GIS QA, including:
    - “Top of perforations” often recorded as the bottom-most screening interval rather than the top-most
    - Static water level and other fields off by up to hundreds of feet
    - Incorporate these types of checks directly in the extraction scripts to reduce reliance on later GIS QA which can be time intensive
    - When conflicts are detected, use AI-extracted data as a replacement if it falls within expected value ranges relative to other related fields
- Additional QA checks to ensure fuzzy-matched low-score addresses remain within previously identified PLSS grids.
- Incorporate confidence scores in JSON outputs to guide QA and validation.

APN, Address, and Location Improvements

- Integrate historical APN datasets to account for subdivisions or APNs that have changed over time.
- Expand location-improvement logic, including using owner name information to fuzzy match against parcel databases and verify whether a corresponding parcel address aligns with the WCR-reported street/location.

- Add logic to split extracted TRS values into Township, Range, and Section fields for new or supplemented records.

Additional Functional Enhancements

- Utilize unredacted reports (when available) to access owner information, which is essential when WCRs list the well location as "same" as the owner's address.

Conclusion

The Well Layer Update project demonstrates the effectiveness of automated data processing and GIS integration in modernizing well record management, as well as the transformative potential of AI-powered OCR software. Recent advancements in AI OCR have significantly improved accuracy—particularly when handling complex layouts, mixed formatting, and handwritten text—making it possible to rapidly digitize and analyze large volumes of historical documentation.

By processing 17,672 well completion reports in under a day, identifying missing records, supplementing key fields, and achieving 91% geocoding accuracy (tied to locations other than PLSS centroids), the project illustrates major gains in efficiency, completeness, and spatial reliability. These outcomes highlight the value of applying AI OCR to historical records across other County divisions, as well as in other counties or organizations with large archival datasets.

As AI technology continues to advance, extraction accuracy will further improve, expanding opportunities to modernize legacy datasets and integrate them into contemporary water management and planning workflows.

The updated well layer is scheduled to be integrated into GISWeb in January 2026, supporting drought resilience and groundwater sustainability efforts, and enhancing water resource decision-making. The project demonstrates how modern tools can transform previously inaccessible or labor-intensive data into a reliable, comprehensive resource for staff, analysts, and policymakers.

Appendix

Below is the script flowchart workflow as well as the primary scripts, with comments explaining the purpose of key functions.

Script Sequence Workflow

C:\Well Layer Update\Python\Azure\OCR Extraction Script secure.py

```python
import os
import json
import fitz
import tempfile
from azure.core.credentials import AzureKeyCredential
from azure.ai.formrecognizer import DocumentAnalysisClient

# Azure config
# Set these in your environment before running the script:
#   AZURE_FORMRECOGNIZER_ENDPOINT = https://your-resource.cognitiveservices.azure.com/
#   AZURE_FORMRECOGNIZER_KEY      = your-key-here


endpoint = os.getenv("AZURE_FORMRECOGNIZER_ENDPOINT")
key      = os.getenv("AZURE_FORMRECOGNIZER_KEY")
model_id = os.getenv("AZURE_FORMRECOGNIZER_MODEL_ID", "wcr11")

# Validate that required variables are present
if not endpoint or not key:
    raise EnvironmentError(
        "Please set the environment variables before running this script."
    )

# Trim any accidental whitespace or trailing slashes
endpoint = endpoint.rstrip("/")

BASE_DIR = r"<internal_project_directory>"

input_folder = f"{BASE_DIR}\WCR_Files"
output_folder = f"{BASE_DIR}\json files"
os.makedirs(output_folder, exist_ok=True)

# Azure client
client = DocumentAnalysisClient(endpoint=endpoint, credential=AzureKeyCredential(key))

#  Compress large image only PDFs
def compress_pdf_with_pymupdf(file_path, max_size_mb=4):
    file_size_mb = os.path.getsize(file_path) / (1024 * 1024)
    is_image_only = True

    try:
        doc = fitz.open(file_path)
        for page in doc:
            if page.get_text().strip():
                is_image_only = False
                break

        if file_size_mb <= max_size_mb or not is_image_only:
```

```
            return file_path  # Skip compression

        print(f"🔧 Compressing large image-only PDF: {os.path.basename(file_path)}")
        temp_file = tempfile.NamedTemporaryFile(delete=False, suffix=".pdf")
        temp_file.close()  # Close so fitz can write to it
        new_doc = fitz.open()

        for page in doc:
            pix = page.get_pixmap(dpi=100)
            img_bytes = pix.tobytes("jpeg")
            img_doc = fitz.open("jpeg", img_bytes)
            rect = img_doc[0].rect
            pdf_page = new_doc.new_page(width=rect.width, height=rect.height)
            pdf_page.insert_image(rect, stream=img_bytes)

        new_doc.save(temp_file.name)
        new_doc.close()
        return temp_file.name

    except Exception as e:
        print(f"Could not compress PDF ({file_path}): {e}")
        return file_path

# Process each PDF
for dirpath, dirnames, filenames in os.walk(input_folder):
    for filename in filenames:  #iterate over the filenames list
        if filename.lower().endswith(".pdf"):
            original_path = os.path.join(dirpath, filename)  # Use dirpath to construct full
path
            pdf_to_process = compress_pdf_with_pymupdf(original_path)

            print(f"\n📄 Processing file: {filename}")

            with open(pdf_to_process, "rb") as f:
                try:
                    poller = client.begin_analyze_document(
                        model_id=model_id,
                        document=f
                    )
                    result = poller.result()

                    result_dict = result.to_dict()
                    json_filename = os.path.splitext(filename)[0] + ".json"
                    json_path = os.path.join(output_folder, json_filename)

                    with open(json_path, "w", encoding="utf-8") as json_file:
                        json.dump(result_dict, json_file, indent=2)

                    print(f"Saved JSON: {json_path}")

```

```
                except Exception as e:
                    print(f"Error processing {filename}: {e}")

            # Clean up temp file if we compressed
            if pdf_to_process != original_path:
                try:
                    os.remove(pdf_to_process)
                    print(f"Deleted temporary file: {os.path.basename(pdf_to_process)}")
                except Exception as e:
                    print(f"Failed to delete temp file: {e}")
```

C:\Well Layer Update\Python\Azure\Data Extraction and Classification Script.py

```python
import os
import json
import pandas as pd
from fuzzywuzzy import fuzz
import re

BASE_DIR = r"<internal_project_directory>"

# Paths for input JSON files and output Excel file
input_folder = f"{BASE_DIR}\json files"
output_excel = f"{BASE_DIR}\extracted_data_dwr.xlsx"

# Destruction related keywords (including common misspellings)
destruction_keywords = [
    "abandonment", "abandon", "abandone", "abandoment",
    "destroy", "destruction", "distruction", "demolish", "decommision", "decommission", "destroying"
]
destruction_keywords = [kw.lower() for kw in destruction_keywords]

# Extract raw text from JSON content for keyword search
def extract_raw_text(json_content):
    # Try to get the main content field
    raw_text = json_content.get("content", "").lower()
    if not raw_text:  # Fallback to concatenating line content from pages
        raw_text = " ".join(
            line["content"].lower()
            for page in json_content.get("pages", [])
            for line in page.get("lines", [])
            if "content" in line and isinstance(line["content"], str)
        )
    return raw_text

#Count destruction keywords using fuzzy matching to handle OCR errors
def count_destruction_keywords(text, keywords, threshold=85):
    words = re.findall(r'\b\w+\b', text)  # Split into words
    matched = []
    for word in words:
        scores = [fuzz.ratio(word, kw) for kw in keywords]
        max_score = max(scores, default=0)
        if max_score >= threshold:
            best_index = scores.index(max_score)
            best_match = keywords[best_index]
            matched.append((word, best_match))
    # Get unique original words
    unique_matched = sorted(set(word for word, _ in matched))
    return unique_matched

```

```
# Initialize lists for data storage
main_data = []
casings_data = []

# Process JSON files to extract fields, generate destruction keywords, and structure data
for filename in os.listdir(input_folder):
    if filename.endswith(".json"):
        path = os.path.join(input_folder, filename)
        with open(path, "r", encoding="utf-8") as file:
            try:
                json_content = json.load(file)
                doc = json_content.get("documents", [{}])[0]
                fields = doc.get("fields", {})

                # Extract raw text for keyword search
                raw_text = extract_raw_text(json_content)
                matched_keywords = count_destruction_keywords(raw_text, destruction_keywords)

                # Build keyword string
                keywords_str = ', '.join(matched_keywords) if matched_keywords else ''

                # Main data: all top-level fields except Casings and Log, preserve existing destruction_well
                record = {
                    k: (v.get("content") if isinstance(v, dict) else v)
                    for k, v in fields.items()
                    if k not in ["Log", "Casings"]
                }
                record["source_file"] = filename
                record["additional_destruction_keywords"] = keywords_str

                main_data.append(record)

                # Casings data: parse each entry in the list
                if "Casings" in fields and fields["Casings"].get("value_type") == "list":
                    for idx, casing in enumerate(fields["Casings"]["value"], start=1):
                        row = {
                            "source_file": filename,
                            "row_index": idx
                        }
                        if casing.get("value_type") == "dictionary":
                            for key, val in casing["value"].items():
                                row[key] = val.get("content") if isinstance(val, dict) else val
                            casings_data.append(row)

            except Exception as e:
                print(f"Error processing {filename}: {e}")

# write extracted and classified data to Excel with separate sheets for Main Data and Casings
```

```
96  with pd.ExcelWriter(output_excel, engine="openpyxl") as writer:
97      pd.DataFrame(main_data).to_excel(writer, index=False, sheet_name="Main Data")
98      pd.DataFrame(casings_data).to_excel(writer, index=False, sheet_name="Casings")
99
100 print(f"Excel file created with 'Main Data' and 'Casings' at: {output_excel}")
```

C:\Well Layer Update\Python\Azure\Data Cleaning and Matching Script.py

```python
import pandas as pd
import re
from rapidfuzz import fuzz, process

BASE_DIR = r"<internal_project_directory>"
# === File Paths ===
input_excel = f"{BASE_DIR}\extracted_data_SCC_3.xlsx"
parcel_excel = f"{BASE_DIR}\Parcels.xlsx"
output_excel = f"{BASE_DIR}\OCR(SCC1)_3.xlsx"
unique_excel = f"{BASE_DIR}\unique_values.xlsx"

def clean_well_id(raw):
    if pd.isna(raw):
        return ""

    original = str(raw).strip()
    # Convert 'c' or 'C' to 'e' at the start
    if original.lower().startswith('c'):
        original = 'e' + original[1:]
    original = re.sub(r'[oO]', '0', original)
    cleaned = original.replace(" ", "").upper().strip()

    # Handle well IDs starting with E/e
    if original.lower().startswith('e'):
        temp = original.lower().replace('o', '0').upper()
        match = re.match(r"^(E)(\d{7,8})(.*)$", temp)
        if match:
            prefix = match.group(1)
            digits = match.group(2)
            return prefix + digits
        digits_after_e = re.sub(r"\D", "", temp[1:])
        if 6 <= len(digits_after_e) <= 8:
            return "E" + digits_after_e
        digits_match = re.match(r"^(E)(\d+)", temp)
        if digits_match:
            return digits_match.group(1) + digits_match.group(2)
        return temp

    # Preserve WCR-formatted IDs
    if re.match(r"^WCR\d{4}-\d{6}$", cleaned):
        return cleaned

    # Fallback cleanup for non-WCR values
    text = original.lower().replace(",", "").replace("no.", "").replace("/", " ")
    text = re.sub(r"[\s\n\r]+", " ", text).strip()

    # Check for date-like prefix (e.g., MM-DD-YYYY or MM.DD.YYYY)
    match = re.match(r"^(\d{1,2})[\.-](\d{1,2})[\.-](\d{2,4})(\d{5,7})$", text)
```

```
    if match:
        year_str = match.group(3)
        try:
            year = int(year_str)
            if year < 100:
                year += 1900 if year >= 35 else 2000
            if 1935 <= year <= 2025:
                return match.group(4)
        except ValueError:
            pass

    # Remove date-like prefix if present
    prefix_removed = re.sub(r"^\d{1,2}[\.-]\d{1,2}[\.-]\d{2,4}", "", text).strip()
    segments = re.split(r"[\s]+", prefix_removed)

    #Collect all digit sequences (5-8 digits) from segments
    all_segment_digits = []
    for segment in segments:
        segment_digits = re.findall(r"\d{5,8}", segment)
        all_segment_digits.extend(segment_digits)

    if all_segment_digits:
        # If multiple sequences, prefer the last one unless an earlier one i longer
        longest = max(all_segment_digits, key=len)
        last_segment_digits = [d for d in re.findall(r"\d{5,8}", segments[-1])] if segments else []
        if last_segment_digits and len(max(last_segment_digits, key=len, default="")) >= len(longest) - 1:
            return max(last_segment_digits, key=len)
        return longest

    # Global fallback for any remaining digit sequences
    digits = re.findall(r"\d{5,8}", text)
    if digits:
        return max(digits, key=len)

    return ""

# Load OCR Extracted Data
df = pd.read_excel(input_excel, dtype=str).fillna("")

# Load unique values for fuzzy matching
unique_df = pd.read_excel(unique_excel, header=None)
field_names = ['city', 'driller_name', 'drilling_method', 'fluid', 'test_type']
uniques_dict = {}
for idx, field in enumerate(field_names):
    col_data = unique_df[idx].dropna().tolist()
    if col_data:
        uniques_dict[field] = col_data[1:]  # Skip the first row (label)

```

```python
#Map to df column names (lowercase as per input)
col_map = {
    'city': 'city',
    'driller_name': 'contractor',
    'drilling_method': 'drilling_method',
    'fluid': 'fluid',
    'test_type': 'test_type'
}

# Clean each field using fuzzy matching to unique values, appending score for non-exact matches or '(no match)' if no match found
for field, col in col_map.items():
    if col in df.columns:
        def clean_val(val):
            if pd.isna(val) or not str(val).strip():
                return val
            val_str = str(val).strip().lower()
            choices = uniques_dict.get(field, [])
            if not choices:
                return val_str
            choices_lower = [str(c).strip().lower() for c in choices]
            choices_map = dict(zip(choices_lower, choices))
            if val_str in choices_lower:
                return choices_map[val_str]
            result = process.extractOne(val_str, choices_lower, scorer=fuzz.partial_ratio)
            if result:
                match_lower, score, _ = result
                if score >= 70:
                    return f"{choices_map.get(match_lower, match_lower)} (score={score})"
            return f"{val_str} (no match)"
        df[col] = df[col].apply(clean_val)

# Column names
APN_COL = "apn"
ADDR_COL = "address"
FALLBACK_ADDR_COL = "owner_address"
LOC_DESC_COL = "location_details"
TEMPLATE_ID_COL = "revision_cleaned"

# Define template signatures
TEMPLATE_SIGNATURES = {
    "Template 1": ['1197', '790', '0503'],
    "Template 2": ['965', '905', '968', '958', '969', '960', '246354', '354', '554', '154', '334', '384', '954', '304', '254', '454', '568', '934', '87649354', '87028354'],
    "Template 3": ['776', '1286', '1288', '775', '778', '7741'],
    "Template 4": ['46170', '46370'],
    "Template 5": ['115', '118', '113', '119'],
    "Template 6": ['246', '2467'],
    "Template 7": ['263', '253', '2634764'],
    "Template 8": ['188'],
```

```
    "Template 9": ['1200', '12008', '112006', '12006', '1200', '12000', '12009', '10008', '40008']
}

DATE_SIGNATURES = {
    "Template 8": ['12/19/2017'],
    "Template 9": ['1/2006', '1/2008', '1/2000', '1/2005']
}

flat_signature_map = [(val, template) for template, vals in TEMPLATE_SIGNATURES.items() for val in vals]

flat_date_signature_map = [(val, template) for template, vals in DATE_SIGNATURES.items() for val in vals]

def preprocess_apn(apn):
    apn = str(apn).strip()
    if '\n' in apn:
        parts = [p.strip() for p in apn.split('\n') if p.strip()]
        if len(parts) == 3 and all(p.isdigit() for p in parts):
            first, second, third = parts
            if len(third) > len(second):
                second, third = third, second
            return '-'.join([first, second, third])
    return apn

def clean_apn_format(apn):
    if not isinstance(apn, str):
        return ""

    apn = preprocess_apn(apn)

    # Remove all non-digit characters
    digits = re.sub(r"\D", "", apn)

    # Expecting 8 digits (3+3+2 format)
    if len(digits) == 8:
        return f"{digits[:3]}-{digits[3:6]}-{digits[6:]}"

    # Sometimes 7-digit (e.g., missing leading 0 in first section)
    elif len(digits) == 7:
        return f"{digits[:2].zfill(3)}-{digits[2:5]}-{digits[5:].zfill(2)}"

    # Other cases invalid
    return ""

def clean_revision(value):
    if pd.isna(value) or not str(value).strip():
        return ""
    value = str(value).strip()
    # Check for date-like patterns
```

```
    date_patterns = [
        r"^\d{1,2}/\d{1,2}/\d{4}$",  # MM/DD/YYYY or M/D/YYYY
        r"^\d{1,2}-\d{1,2}-\d{4}$",  # MM-DD-YYYY or M-D-YYYY
        r"^\d{1,2}/\d{4}$",          # M/YYYY or MM/YYYY
        r"^\d{1,2}-\d{4}$"           # M-YYYY or MM-YYYY
    ]
    for pattern in date_patterns:
        if re.match(pattern, value):
            return value  # Return original string if it's a date
    # Otherwise, strip all non-digit characters and newlines
    digits = re.sub(r"\D", "", value)
    return digits

def identify_template(row):
    text = row.get(TEMPLATE_ID_COL, "")
    if not isinstance(text, str) or not text.strip():
        return "Unknown"
    # First check for date signatures (exact match)
    for val, template in flat_date_signature_map:
        if text == val:
            return template
    # Then check for numeric signatures (fuzzy match)
    match, score, _ = process.extractOne(
        text.strip(), [val for val, _ in flat_signature_map], scorer=fuzz.token_sort_ratio
    )
    if score >= 80:
        for val, template in flat_signature_map:
            if match == val:
                return template
    return "Unknown"

#Create revision_cleaned column before template identification
df["revision_cleaned"] = df["revision"].apply(clean_revision)
df["template_id"] = df.apply(identify_template, axis=1)

# Known baseline occurrences per template
known_baselines = {
    "Template 1": 1,  # DESTROY- (1)
    "Template 2": 2,  # destruction- (1) + Destroying- (1)
    "Template 3": 2,  # Destruction (1) + destruction- (1)
    "Template 6": 1,  # abandoned (1)
    "Template 9": 1,  # DESTROY- (1)
    "Template 4": 0,
    "Template 5": 0,
    "Template 7": 0,
    "Template 8": 0,
    "Unknown": 0
}

# Baseline keywords per template (lowercase)
```

```
baseline_keywords = {
    "Template 1": ['destroy'],
    "Template 2": ['destruction', 'destroying'],
    "Template 3": ['destruction'],
    "Template 6": ['abandoned'],
    "Template 9": ['destroy'],
    "Template 4": [],
    "Template 5": [],
    "Template 7": [],
    "Template 8": [],
    "Unknown": []
}

# Function to process additional_destruction_keywords and set keyword_destruction_well
def enhance_destruction_detection(row):
    template = row['template_id']
    keywords_str = row.get('additional_destruction_keywords', '')
    baseline_set = set(baseline_keywords.get(template, []))
    destruction_well = row.get('destruction_well', ':unselected:').strip()

    if not keywords_str:
        row['additional_destruction_keywords'] = ''
        row['keyword_destruction_well'] = destruction_well
        return row

    # Split into list, strip spaces and lowercase for comparison
    keyword_list = [kw.strip().lower() for kw in keywords_str.split(',')]

    # Filter out 'instruction' or items starting with 'instruction'
    filtered_list = [kw for kw in keyword_list if not kw.startswith('instruction')]

    # Mark keywords based on template
    marked_list = []
    if template == "Unknown":
        marked_list = [f"{kw} (unknown)" for kw in filtered_list]
    else:
        for kw in filtered_list:
            if kw in baseline_set:
                marked_list.append(f"{kw} (baseline)")
            else:
                marked_list.append(kw)

    # Update the column to cleaned/marked string
    row['additional_destruction_keywords'] = ', '.join(marked_list)

    # Set keyword_destruction_well: default to destruction_well
    row['keyword_destruction_well'] = destruction_well

    # For known templates, override to :selected: if keyword count exceeds baseline and
destruction_well is :unselected:
```

```
    if template != "Unknown":
        total_keyword_count = len(filtered_list)
        baseline_count = known_baselines.get(template, 0)
        if total_keyword_count > baseline_count and destruction_well == ":unselected:":
            row['keyword_destruction_well'] = ':selected:'
            # Unset other fields
            for field in ['deepen_well', 'recondition_other.', 'domestic_well', 'irrigation_well',
                          'public_well', 'monitoring_well', 'industrial_well', 'other_well',
                          'new_well', 'test_well']:
                row[field] = ':unselected:'

    return row

# Apply the enhancement after template_id is set
df = df.apply(enhance_destruction_detection, axis=1)

# Normalize other_well_no for Template 6 and merge duplicates
def normalize_other_well_no(val):
    if not isinstance(val, str):
        return ""
    return re.sub(r"[^\w]", "", val).upper()  # Remove all non-alphanumeric characters and uppercase

# Only process Template 6
template6_df = df[df["template_id"] == "Template 6"].copy()
template6_df["other_well_no_normalized"] = template6_df["other_well_no"].apply(normalize_other_well_no)

# Group by normalized other_well_no and merge fields
merged_template6 = (
    template6_df.groupby("other_well_no_normalized", as_index=False)
    .agg(lambda x: next((i for i in x if i.strip()), "") if x.dtype == "O" else x.iloc[0])
)

# Drop duplicates in original df
df = df[df["template_id"] != "Template 6"]

# Append merged template6 records back
df = pd.concat([df, merged_template6], ignore_index=True)

# Load Casings Sheet
casings_df = pd.read_excel(input_excel, sheet_name="Casings", dtype=str).fillna("")
casings_df["row_index"] = casings_df["row_index"].astype(int)
casings_df["top"] = casings_df["top"].astype(str)
casings_df["bottom"] = casings_df["bottom"].astype(str)

def clean_to_digits(value, is_top=False):
    if pd.isna(value) or not str(value).strip():
        return None
    # Extract the first sequence of digits before a non-digit character
```

```
    match = re.match(r"^\d+", str(value).strip())
    if match:
        cleaned = match.group(0)
        if not cleaned.isdigit():
            return None
        result = int(cleaned)
        # For top values > 999 ending with '0', strip the trailing '0'
        if is_top and result > 999 and cleaned.endswith('0'):
            result = int(cleaned[:-1])  # Remove the last digit ('0')
        return result
    return None

def extract_perforation_fields(row):
    template = row.get("template_id")
    source_file = row.get("source_file", "").strip()

    if template == "Template 1":
        selected_rows = []
        for i in range(1, 7):
            colname = f"template1_screen_row{i}"
            if row.get(colname, "").strip() == ":selected:":
                selected_rows.append(i)

        if not selected_rows:
            return pd.Series({"perforation_interval": "", "top_perforations": "",
"bottom_perforations": ""})

        matches = casings_df[
            (casings_df["source_file"] == source_file) &
            (casings_df["row_index"].isin(selected_rows))
        ].copy()

        if matches.empty:
            return pd.Series({"perforation_interval": "", "top_perforations": "",
"bottom_perforations": ""})

        matches["top_cleaned"] = matches["top"].apply(lambda x: clean_to_digits(x,
is_top=True))
        matches["bottom_cleaned"] = matches["bottom"].apply(clean_to_digits)

        top = matches["top_cleaned"].min()
        bottom = matches["bottom_cleaned"].max()

        if pd.isna(top) or pd.isna(bottom):
            return pd.Series({"perforation_interval": "", "top_perforations": "",
"bottom_perforations": ""})

        # Check if bottom is less than top
        if bottom < top:
            return pd.Series({
                "perforation_interval": "",
```

```
                "top_perforations": top,
                "bottom_perforations": ""
            })

        return pd.Series({
            "perforation_interval": f"{top} - {bottom}",
            "top_perforations": top,
            "bottom_perforations": bottom
        })

    elif template in {"Template 2", "Template 3", "Template 4", "Template 6", "Template 8", "Template 9", "Unknown"}:
        filtered = casings_df[casings_df["source_file"] == source_file].copy()

        if template in {"Template 8", "Template 9"}:
            if template == "Template 8":
                has_empty_type = (filtered["type"] == "").any()
                if has_empty_type:
                    # Misidentified Template 8: process all rows (like Templates 2, 3, 4, 6, Unknown)
                    pass
                else:
                    # True Template 8: only "Screen" rows
                    filtered = filtered[filtered["type"] == "Screen"]
            elif template == "Template 9":
                filtered = filtered[filtered["type"] == "Screen"]

        if filtered.empty:
            return pd.Series({"perforation_interval": "", "top_perforations": "", "bottom_perforations": ""})

        filtered["top_cleaned"] = filtered["top"].apply(lambda x: clean_to_digits(x, is_top=True))
        filtered["bottom_cleaned"] = filtered["bottom"].apply(clean_to_digits)

        filtered = filtered[
            (filtered["top_cleaned"].notnull()) &
            (filtered["bottom_cleaned"].notnull()) &
            (filtered["bottom_cleaned"] < 2000)
        ]

        if filtered.empty:
            return pd.Series({"perforation_interval": "", "top_perforations": "", "bottom_perforations": ""})

        top_row = filtered.loc[filtered["row_index"].idxmin()]
        bottom_row = filtered.loc[filtered["row_index"].idxmax()]

        top = top_row["top_cleaned"]
        bottom = bottom_row["bottom_cleaned"]

```

```
        if top is None or bottom is None:
            return pd.Series({"perforation_interval": "", "top_perforations": "", "bottom_perforations": ""})

        # Check if bottom is less than top
        if bottom < top:
            return pd.Series({
                "perforation_interval": "",
                "top_perforations": top,
                "bottom_perforations": ""
            })

        return pd.Series({
            "perforation_interval": f"{top} - {bottom}",
            "top_perforations": top,
            "bottom_perforations": bottom
        })

    elif template == "Template 5":
        raw_text = row.get("perforations_template5", "")
        if not raw_text.strip():
            return pd.Series({"perforation_interval": "", "top_perforations": "", "bottom_perforations": ""})

        ranges = re.findall(r"(\d{2,4})\s*[--]\s*(\d{2,4})", raw_text)
        numbers = []

        for start, end in ranges:
            try:
                start_int = int(start)
                end_int = int(end)
                if 10 < start_int < 2000 and 10 < end_int < 2000:
                    numbers.extend([start_int, end_int])
            except ValueError:
                continue

        singles = re.findall(r"\b(\d{2,4})\b", raw_text)
        for val in singles:
            try:
                num = int(val)
                if 10 < num < 2000:
                    numbers.append(num)
            except ValueError:
                continue

        if numbers:
            top = min(numbers)
            bottom = max(numbers)
            # Check if bottom is less than top
            if bottom < top:
```

```
                return pd.Series({
                    "perforation_interval": "",
                    "top_perforations": top,
                    "bottom_perforations": ""
                })
            return pd.Series({
                "perforation_interval": f"{top} - {bottom}",
                "top_perforations": top,
                "bottom_perforations": bottom
            })

        return pd.Series({"perforation_interval": "", "top_perforations": "", "bottom_perforations": ""})

    elif template == "Template 7":
        raw_text = row.get("perforations_template7", "")
        if not raw_text.strip():
            return pd.Series({"perforation_interval": "", "top_perforations": "", "bottom_perforations": ""})

        ranges = re.findall(r"(\d{2,4})\s*[--]\s*(\d{2,4})", raw_text)
        singles = re.findall(r"\b(\d{2,4})\b", raw_text)
        numbers = []

        for start, end in ranges:
            try:
                start_int = int(start)
                end_int = int(end)
                if 10 < start_int < 2000 and 10 < end_int < 2000:
                    numbers.extend([start_int, end_int])
            except ValueError:
                continue

        for val in singles:
            try:
                num = int(val)
                if 10 < num < 2000:
                    numbers.append(num)
            except ValueError:
                continue

        if numbers:
            top = min(numbers)
            bottom = max(numbers)
            # Check if bottom is less than top
            if bottom < top:
                return pd.Series({
                    "perforation_interval": "",
                    "top_perforations": top,
                    "bottom_perforations": ""
```

```
                })
            return pd.Series({
                "perforation_interval": f"{top} - {bottom}",
                "top_perforations": top,
                "bottom_perforations": bottom
            })

        return pd.Series({"perforation_interval": "", "top_perforations": "", "bottom_perforations": ""})

    return pd.Series({"perforation_interval": "", "top_perforations": "", "bottom_perforations": ""})

df[["perforation_interval", "top_perforations", "bottom_perforations"]] = df.apply(extract_perforation_fields, axis=1)

# Clean Well ID
df["well_id_cleaned"] = df.iloc[:, 0].apply(clean_well_id)

# Extract potential APN pattern from a text string
def extract_apn_from_text(text):
    if not isinstance(text, str):
        return ""
    text = text.upper().strip()

    text = text.replace("\n", " ").replace("APN#", "APN").replace("A.P.N.", "APN")

    match = re.search(r"\b(\d{2,3}-\d{3}-\d{2}[A-Z]?)\b", text)
    if match:
        return match.group(1)
    return ""

def fallback_apn_with_source(row):
    apn_raw = str(row.get(APN_COL, "")).strip()
    apn_digits = clean_apn_format(apn_raw)

    if apn_digits:
        return apn_digits.zfill(7), "Original APN field"

    apn_candidate = extract_apn_from_text(row.get("location_details", ""))
    source = "location_details"

    if not apn_candidate:
        apn_candidate = extract_apn_from_text(row.get("owner_well_id", ""))
        source = "owner_well_id"

    if apn_candidate:
        digits_only = clean_apn_format(apn_candidate)
        return digits_only.zfill(7), f"Extracted from {source}"

    return "", "No APN found"
```

```
577
578 df[["apn_cleaned", "apn_cleaned_source"]] = df.apply(fallback_apn_with_source, axis=1,
    result_type="expand")
579
580 # Define cleaning function for pump_test_length
581 def clean_pump_test_length(value):
582     if pd.isna(value) or not str(value).strip():
583         return ""
584
585     value = str(value).strip()
586
587     # Return empty string if slash is present
588     if '/' in value:
589         return ""
590
591     # Remove any digits (and subsequent characters) after a non-digit, non-period character
592     match = re.match(r'^[\d.]+', value)
593     if not match:
594         return ""
595
596     cleaned = match.group(0)
597
598     # Keep only digits and periods
599     cleaned = re.sub(r'[^0-9.]', '', cleaned)
600
601     # Ensure valid number format (remove multiple periods, trailing/leading periods)
602     if cleaned.count('.') > 1:
603         parts = cleaned.split('.')
604         cleaned = parts[0] + '.' + ''.join(parts[1:])
605     cleaned = cleaned.strip('.')
606
607     # Return empty string if result is empty or invalid
608     return cleaned if cleaned and cleaned != '.' else ""
609
610 numeric_columns = {
611     "water_static": "water_static_cleaned",
612     "depth_well": "depth_well_cleaned",
613     "yield": "yield_cleaned",
614     "seal": "seal_cleaned",
615     "first_water": "first_water_cleaned",
616     "total_depth": "total_depth_cleaned",
617     "lat": "lat_cleaned",
618     "long": "long_cleaned",
619     "pump_test_length": "pump_test_length_cleaned"
620 }
621
622 def clean_numeric_field(value):
623     if not isinstance(value, str):
624         value = str(value)
625
```

```
    value = value.strip().lower()

    # Handle newline: keep only the first number before the newline
    if '\n' in value:
        value = value.split('\n')[0].strip()

    # Remove apostrophe and any digits following it
    value = re.sub(r"'\s*\d*(\"|''|’)?", "", value)

    value = re.sub(r'^[^\d]+', '', value)      #Remove non-digit prefix
    value = re.sub(r'[^\d]+$', '', value)

    range_pattern = r"^\s*(\d+(?:\.\d+)?)\s*(?:[--]|to|t)\s*(\d+(?:\.\d+)?)\s*$"
    range_match = re.match(range_pattern, value)
    if range_match:
        try:
            num1 = float(range_match.group(1))
            num2 = float(range_match.group(2))
            avg = (num1 + num2) / 2
            return f"{avg:.2f}".rstrip('0').rstrip('.')
        except ValueError:
            pass

    value = re.sub(r'(?<=\d),(?=\d)', '.', value)
    value = re.sub(r'[^0-9.]', '', value)

    if value.count('.') > 1:
        parts = value.split('.')
        value = parts[0] + '.' + ''.join(parts[1:])

    return value

for colname, new_col in numeric_columns.items():
    if colname == "pump_test_length":
        df[new_col] = df[colname].apply(clean_pump_test_length)
    else:
        df[new_col] = df[colname].apply(clean_numeric_field)


# Initialize qa_flag column
df['depths_flag'] = ''

# Only check rows where both fields are non-empty
mask = (df['total_depth_cleaned'] != '') & (df['depth_well_cleaned'] != '')

# Apply logic: if total_depth < depth_well → flag 'review depths'
df.loc[mask, 'depths_flag'] = df.loc[mask].apply(
    lambda row: 'review depths' if float(row['total_depth_cleaned']) <
float(row['depth_well_cleaned']) else '',
    axis=1
```

```
)


# Load and structure alias dictionary
alias_df = pd.read_excel(r"C:\Well Layer Update\GIS\data_tables\street_names_with_aliases.xlsx").fillna("")
alias_dict = {}
for _, row in alias_df.iterrows():
    canonical = row['canonical_name'].strip().upper()
    for alias in str(row['alias']).split(";"):
        alias = alias.strip().upper()
        if alias:
            alias_dict[alias] = canonical

for canonical in alias_df["canonical_name"].unique():
    canonical = canonical.strip().upper()
    alias_dict[canonical] = canonical

street_keywords = set(alias_dict.keys())

def extract_address_from_location(desc):
    if not isinstance(desc, str):
        return ""
    desc = desc.upper()
    desc = re.sub(r"[^\w\s]", " ", desc)
    desc = re.sub(r"\s+", " ", desc).strip()

    if re.search(r"\b(UTM|TOPO|SHEET|1[:;,\.]?62[,\.]?500|GRID|QUAD|COORDINATE|LAT|LONG|MAP)\b", desc):
        return ""

    if re.search(r"\b\d{1,5}\s*(FEET|FT|YARDS|MILES?)\b", desc):
        return ""

    tokens = desc.split()
    num_tokens = len(tokens)

    for i in range(num_tokens):
        for j in range(1, 5):
            if i + j > num_tokens:
                continue
            candidate_tokens = tokens[i:i + j]
            candidate = " ".join(candidate_tokens)
            if candidate in street_keywords:
                canonical = alias_dict.get(candidate, candidate)
                if i > 0 and re.fullmatch(r"\d{2,6}", tokens[i - 1]):
                    return f"{tokens[i - 1]} {canonical}"
                else:
                    return canonical

```

```
    for i in range(num_tokens):
        for j in range(1, 5):
            if i + j > num_tokens:
                continue
            candidate_tokens = tokens[i:i + j]
            candidate = " ".join(candidate_tokens)
            best_match, score, _ = process.extractOne(
                candidate,
                list(street_keywords),
                scorer=fuzz.token_sort_ratio
            )
            if score >= 85:
                canonical = alias_dict.get(best_match, best_match)
                if i > 0 and re.fullmatch(r"\d{2,6}", tokens[i - 1]):
                    return f"{tokens[i - 1]} {canonical}"
                else:
                    return canonical

    return ""

def resolve_address_with_source(row):
    addr = str(row.get(ADDR_COL, "")).strip()
    fallback = str(row.get(FALLBACK_ADDR_COL, "")).strip()
    loc_extracted = extract_address_from_location(row.get(LOC_DESC_COL, ""))

    fallback_keywords = ["same", "dame", "come", "came", "some", "sama", "same as above"]
    if any(fuzz.ratio(addr.lower(), word) >= 80 for word in fallback_keywords):
        return fallback, "Fallback address (approximate 'same')"

    if addr:
        return addr, "Direct address"
    if loc_extracted:
        return loc_extracted, "Extracted from location description"
    if fallback:
        return fallback, "Fallback address (owners)"
    return "", "No address available"

def strip_city_and_zip(address):
    if not address:
        return address

    cities = [
        "APTOS", "BEN LOMOND", "BOULDER CREEK", "CAPITOLA", "DAVENPORT",
        "LOS GATOS", "SANTA CRUZ", "SCOTTS VALLEY", "SOQUEL", "WATSONVILLE"
    ]
    address = address.upper()

    house_number_match = re.match(r"^\d+\s+", address)
    if not house_number_match:
        return address
```

```
773
774     start = house_number_match.end()
775     post_number = address[start:]
776     tokens = post_number.split()
777     retained_tokens = tokens[:1]
778     remaining_tokens = tokens[1:]
779
780     for idx, token in enumerate(remaining_tokens):
781         joined = " ".join([*retained_tokens, *remaining_tokens[:idx + 1]])
782         for city in cities:
783             if city in joined:
784                 return address[:address.upper().find(city)].strip()
785
786     return address
787
788 def clean_for_matching(addr):
789     if not addr:
790         return ""
791     addr = addr.upper()
792     addr = re.sub(r"\s*\n\s*", ", ", addr)
793     addr = re.sub(r"\s+", " ", addr).strip()
794
795
796     remove_keywords = [
797         "P.O. BOX", "PO BOX", "O. BOX", "P.O.", "PO", "BOX", "PABOX", "DRAWER",
798         "ABOVE", "OFF", "MAP", "MAPSHEET", "APN", "NO ADDRESS", "END OF", "ENDOF",
799         "PART OF", "P/R", "SUITE", "C/O", "SHOPS", "NOTA", "WIR", "EX", "FEET NORTH OF"
800     ]
801     for keyword in remove_keywords:
802         addr = re.sub(rf"\b{re.escape(keyword)}\b", "", addr)
803
804     #dictionary and spell-checker for common OCR mispellings
805     ocr_fixes = {
806         r"\bARE\b": "AVE", r"\bSTH\b": "ST", r"\bDRI\b": "DR", r"5T": "ST",
807         r"\bOAK STREE\b": "OAK ST", r"(\d)O(\d)": r"\1 0 \2",
808         r"\bMOUNT\b": "MT", r"\bM\b": "MT", r"\bR\b": "RD",
809         r"\bDRIE\b": "DR", r"\bQUINET\b": "QUINNETTE", r"\bJANE\b": "LANE",
810         r"\bCEUK\b": "CREEEK", r"\bTHNY\b": "TWINY", r"\bELTON\b": "FELTON",
811         r"\bBENLOMOND\b": "BEN LOMOND", r"\bBENLAMIND\b": "BEN LOMOND",
812         r"\bBEN SOMAND\b": "BEN LOMOND", r"\bBEN HOWARD\b": "BEN LOMOND",
813         r"\bBONNY DOON\b": "BONNY DOON", r"\bBUNNY DOON\b": "BONNY DOON",
814         r"\bBROOKDALE\b": "", r"\bSANTACRUZ\b": "SANTA CRUZ",
815         r"\bSANTACURZ\b": "SANTA CRUZ", r"\bCUPERTING\b": "CUPERTINO",
816         r"\bROALND\b": "ROLAND", r"\bZIMT\b": "", r"\bHERMOURA\b": "HERMOSA",
817         r"\bDALTON\b": "", r"\bSA\b": "CA", r"\bCALIFÓRNIA\b": "CA",
818         r"\bCALLIL\b": "CA", r"\bCALIF\b": "CA", r"\bCALL\b": "CA", r"\bCAO\b": "CA",
819         r"\bCAMILLA\b": "CAMPBELL", r"\bLETT?S VALLY\b": "LETTS VALLEY",
820         r"\bENGINE GRADE\b": "EMPIRE GRADE", r"\bFORMAND\b": "LOMOND",
821         r"\bSTREEL\b": "STREET", r"\bARE\b": "AVE", r"\bMª\b": "MA", r"\bª\b": "A"
822     }
```

```
    for pattern, replacement in ocr_fixes.items():
        addr = re.sub(pattern, replacement, addr)

    # Remove irrelevant leading words found in well reports
    addr = re.sub(r"^(WELL( SITE| HEAD| SA| LOCATION| HOUSE)?|ADDRESS|LOCATED AT|NEAR|NEXT TO|ADJACENT TO|BEHIND|ADJOINING)\s+", "", addr)

    # Normalize street suffixes for consistent matching
    replacements = {
        " STREET": " ST", " AVENUE": " AVE", " ROAD": " RD", " DRIVE": " DR",
        " BOULEVARD": " BLVD", " MOUNT ": " MT ", " HIGHWAY": " HWY",
        " LANE": " LN", " COURT": " CT", " PLACE": " PL", " TERRACE": " TER",
        " CIRCLE": " CIR", " TRAIL": " TRL", " CREEK": " CRK", " WAY": " WAY"
    }
    for k, v in replacements.items():
        addr = addr.replace(k, v)

    addr = re.sub(r"\s+[A-Z\s]*\sCA\s\d{5,6}$", "", addr)
    addr = re.sub(r"[^\w\s]+$", "", addr)
    addr = re.sub(r"\b\w\b$", "", addr).strip()
    addr = strip_city_and_zip(addr)
    return addr

df[["address_raw", "address_source"]] = df.apply(lambda row: pd.Series(resolve_address_with_source(row)), axis=1)
df["address_cleaned"] = df["address_raw"].astype(str).apply(lambda x: x.splitlines()[0] if x.strip() else "").str.strip().str.upper()
df["address_cleaned_for_matching"] = df["address_cleaned"].apply(clean_for_matching)

# Load Parcel Data and Prepare Matching Columns
parcels_df = pd.read_excel(parcel_excel, dtype=str).fillna("")
parcels_df.columns = parcels_df.columns.str.lower()
parcels_df["clean_apn"] = parcels_df.iloc[:, 0]
parcels_df["clean_address_parcel"] = parcels_df.iloc[:, 13].astype(str).str.strip().str.upper()
parcels_df["match_ready_address"] = parcels_df["clean_address_parcel"]

apn_lookup = parcels_df.set_index("clean_apn")[["clean_address_parcel"]].to_dict("index")

def fuzzy_address_lookup(addr):
    cleaned_input = clean_for_matching(addr)
    if not cleaned_input:
        return "", "", [], 0

    input_house_num = re.match(r"^(\d+)", cleaned_input)
    input_house_num = input_house_num.group(1) if input_house_num else ""

    matches = process.extract(
        cleaned_input,
        parcels_df["match_ready_address"].tolist(),
        scorer=fuzz.token_sort_ratio,
```

```
        limit=30
    )

    candidates_log = []
    best_row, best_score, best_dist = None, 0, float("inf")

    for match_text, score, idx in matches:
        row = parcels_df.iloc[idx]
        match_addr = row["clean_address_parcel"]
        match_num = re.match(r"^(\d+)", match_addr)
        match_num = match_num.group(1) if match_num else ""
        dist = abs(int(input_house_num) - int(match_num)) if input_house_num and match_num else float("inf")
        candidates_log.append(f"{match_addr} (score={score}, dist={dist})")

        if score > best_score or (score == best_score and dist < best_dist):
            best_score, best_dist = score, dist
            best_row = row

    if best_row is not None:
        return best_row["clean_apn"], best_row["clean_address_parcel"], candidates_log, best_score

    return "", "", candidates_log, 0

# Normalize well_id_cleaned for duplicate comparison (strip leading zeros)
def normalize_well_id_for_comparison(well_id):
    if not isinstance(well_id, str) or not well_id.strip():
        return ""
    # Remove leading zeros, but preserve the rest of the ID (including 'E' prefix if present)
    if well_id.startswith('E'):
        return 'E' + well_id[1:].lstrip('0')
    return well_id.lstrip('0')

# Remove Duplicates Based on well_id_cleaned and multiple fields, including all-empty matches
def count_non_empty(row):
    """Count non-empty, non-whitespace fields in a row."""
    return sum(1 for val in row if isinstance(val, str) and val.strip() != "" or not pd.isna(val))

# Add duplicate details column
df['duplicate_exists'] = "No duplicates"

# Add a column to count non-empty fields for each row
df["non_empty_count"] = df.apply(count_non_empty, axis=1)

# Add a column for normalized well_id_cleaned for duplicate comparison
df["well_id_normalized"] = df["well_id_cleaned"].apply(normalize_well_id_for_comparison)

```

```
# Identify duplicates based on normalized well_id_cleaned and any field match or all fields empty
fields_to_check = [
    "total_depth_cleaned",
    "depth_well_cleaned",
    "water_static_cleaned",
    "yield_cleaned",
    "seal_cleaned",
    "first_water_cleaned",
    "permit_date",
    "permit_no",
    "date",
    "apn_cleaned"
]

# Filter for non-empty well_id_cleaned
non_empty_well_id_mask = df["well_id_cleaned"].notna() & (df["well_id_cleaned"] != "")
duplicates = pd.Series([False] * len(df), index=df.index)
for field in fields_to_check:
    # Mark duplicates where normalized well_id_cleaned and field match, and both are non-empty
    mask = (
        df.duplicated(subset=["well_id_normalized", field], keep=False) &
        (df[field].notna() & (df[field] != "")) &
        non_empty_well_id_mask
    )
    duplicates |= mask

# Add duplicates where all fields are empty for a given normalized well_id_cleaned
all_empty_mask = df[fields_to_check].apply(lambda x: x.isna() | (x == "")).all(axis=1)
all_empty_duplicates = (
    df[all_empty_mask & non_empty_well_id_mask].duplicated(subset=["well_id_normalized"], keep=False)
)
duplicates |= all_empty_duplicates

# Identify duplicate groups for logging
duplicate_groups = df[duplicates & non_empty_well_id_mask][["well_id_normalized", "well_id_cleaned", "total_depth_cleaned", "depth_well_cleaned"]].drop_duplicates()

print(f"Found {len(duplicate_groups)} groups of duplicates based on normalized well_id_cleaned and multiple fields or all fields empty")

# Process each duplicate group
records_to_keep = set()
for _, group in duplicate_groups.iterrows():
    well_id_norm = group["well_id_normalized"]
    total_depth = group["total_depth_cleaned"]
    depth_well = group["depth_well_cleaned"]

```

```
    # Get all records in this duplicate group (matching normalized well_id_cleaned and any field or all fields empty)
    group_records = df[
        (df["well_id_normalized"] == well_id_norm) & (
            (df["total_depth_cleaned"] == total_depth) |
            (df["depth_well_cleaned"] == depth_well) |
            (df["water_static_cleaned"] == df[df["well_id_normalized"] == well_id_norm]["water_static_cleaned"].iloc[0]) |
            (df["yield_cleaned"] == df[df["well_id_normalized"] == well_id_norm]["yield_cleaned"].iloc[0]) |
            (df["seal_cleaned"] == df[df["well_id_normalized"] == well_id_norm]["seal_cleaned"].iloc[0]) |
            (df["first_water_cleaned"] == df[df["well_id_normalized"] == well_id_norm]["first_water_cleaned"].iloc[0]) |
            (df["permit_date"] == df[df["well_id_normalized"] == well_id_norm]["permit_date"].iloc[0]) |
            (df["permit_no"] == df[df["well_id_normalized"] == well_id_norm]["permit_no"].iloc[0]) |
            (df["date"] == df[df["well_id_normalized"] == well_id_norm]["date"].iloc[0]) |
            (df["apn_cleaned"] == df[df["well_id_normalized"] == well_id_norm]["apn_cleaned"].iloc[0]) |
            (
                df[fields_to_check].apply(lambda x: x.isna() | (x == "")).all(axis=1) &
                (df["well_id_normalized"] == well_id_norm)
            )
        ) & non_empty_well_id_mask
    ]

    if len(group_records) > 1:
        # Sort by non_empty_count (descending), then by source_file (prefer newer or cleaner files)
        most_complete = group_records.sort_values(
            by=["non_empty_count", "source_file"], ascending=[False, True]
        ).iloc[0]
        df.at[most_complete.name, 'duplicate_exists'] = "Duplicates deleted (matched depths, water_static, yield, seal, first_water, permit_date, permit_no, date, apn_cleaned, or all fields empty, including leading zero variations)"
        records_to_keep.add(most_complete.name)
    else:
        records_to_keep.add(group_records.index[0])

# Keep non-duplicate records and the most complete record from each duplicate group
non_duplicates = df[~duplicates]
duplicates_to_keep = df.loc[list(records_to_keep)]
df = pd.concat([non_duplicates, duplicates_to_keep], ignore_index=True)

# Ensure records with empty well_id_cleaned are marked as "No duplicates"
df.loc[df["well_id_cleaned"].isna() | (df["well_id_cleaned"] == ""), "duplicate_exists"] = "No duplicates"

# Flag distinct duplicates in the final DataFrame, excluding empty well_id_cleaned
```

```
well_id_counts = df[df["well_id_cleaned"].notna() & (df["well_id_cleaned"] != "")]["well_id_normalized"].value_counts()
duplicate_well_ids = well_id_counts[well_id_counts > 1].index

for well_id_norm in duplicate_well_ids:
    mask = df["well_id_normalized"] == well_id_norm
    for idx in df[mask].index:
        current = df.at[idx, 'duplicate_exists']
        if current == "No duplicates":
            df.at[idx, 'duplicate_exists'] = "Distinct duplicates exist (different depths, water_static, yield, seal, first_water, permit_date, permit_no, date, apn_cleaned, and not all empty, including leading zero variations)"
        elif current == "Duplicates deleted (matched depths, water_static, yield, seal, first_water, permit_date, permit_no, date, apn_cleaned, or all fields empty, including leading zero variations)":
            df.at[idx, 'duplicate_exists'] = "Duplicates deleted (matched depths, water_static, yield, seal, first_water, permit_date, permit_no, date, apn_cleaned, or all fields empty, including leading zero variations) and distinct duplicates exist"

# Drop temporary columns
df = df.drop(columns=["non_empty_count", "well_id_normalized"])

# Verify no duplicates remain
remaining_duplicates = df[
    (df["well_id_cleaned"].notna() & (df["well_id_cleaned"] != "") ) & (
        df.duplicated(subset=["well_id_cleaned", "total_depth_cleaned"], keep=False) |
        df.duplicated(subset=["well_id_cleaned", "depth_well_cleaned"], keep=False) |
        df.duplicated(subset=["well_id_cleaned", "water_static_cleaned"], keep=False) |
        df.duplicated(subset=["well_id_cleaned", "yield_cleaned"], keep=False) |
        df.duplicated(subset=["well_id_cleaned", "seal_cleaned"], keep=False) |
        df.duplicated(subset=["well_id_cleaned", "first_water_cleaned"], keep=False) |
        df.duplicated(subset=["well_id_cleaned", "permit_date"], keep=False) |
        df.duplicated(subset=["well_id_cleaned", "permit_no"], keep=False) |
        df.duplicated(subset=["well_id_cleaned", "date"], keep=False) |
        df.duplicated(subset=["well_id_cleaned", "apn_cleaned"], keep=False) |
        (
            df[fields_to_check].apply(lambda x: x.isna() | (x == "")).all(axis=1) &
            df.duplicated(subset=["well_id_cleaned"], keep=False)
        )
    )
]
if not remaining_duplicates.empty:
    print("flag: some duplicates remain")
    print(remaining_duplicates[["well_id_cleaned", "total_depth_cleaned", "depth_well_cleaned", "water_static_cleaned", "yield_cleaned", "seal_cleaned", "first_water_cleaned", "permit_date", "permit_no", "date", "apn_cleaned", "source_file"]])
else:
    print("No duplicates remain.")

# Fuzzy Matching Loop
final_apn, final_address, match_source, fuzzy_logs = [], [], [], []
```

```
1042
1043 for i, (_, row) in enumerate(df.iterrows()):
1044     print(f"Processing record {i + 1} of {len(df)}")
1045     cleaned_apn = row["apn_cleaned"]
1046     cleaned_addr = row["address_cleaned"]
1047     source = row["address_source"]
1048
1049     if cleaned_apn in apn_lookup:
1050         final_apn.append(cleaned_apn)
1051         final_address.append(apn_lookup[cleaned_apn]["clean_address_parcel"])
1052         match_source.append("APN matched")
1053         fuzzy_logs.append("")
1054     else:
1055         if source == "Extracted from location description":
1056             fuzzy_threshold = 75
1057         elif "Fallback address" in source:
1058             fuzzy_threshold = 85
1059         else:
1060             fuzzy_threshold = 62
1061
1062         fuzzy_apn, fuzzy_addr, log, score = fuzzy_address_lookup(cleaned_addr)
1063         if fuzzy_apn and score >= fuzzy_threshold:
1064             final_apn.append(fuzzy_apn)
1065             final_address.append(fuzzy_addr)
1066             if "Fallback address" in source:
1067                 match_source.append(f"Owners address (fallback) matched (score {score})")
1068             else:
1069                 match_source.append(f"Address matched (score {score})")
1070         else:
1071             final_apn.append(cleaned_apn)
1072             final_address.append("")
1073             match_source.append("No match")
1074         fuzzy_logs.append("; ".join(log))
1075
1076 planned_use_columns = {
1077     "domestic_well": "Domestic Well",
1078     "irrigation_well": "Irrigation Well",
1079     "public_well": "Public Supply Well",
1080     "monitoring_well": "Monitoring Well",
1081     "industrial_well": "Industrial Well",
1082     "other_well": "Other Well",
1083     "test_well": "Test Well"
1084 }
1085
1086 def determine_planned_use(row):
1087     selected = [
1088         label for col, label in planned_use_columns.items()
1089         if row.get(col, "").strip() == ":selected:"
1090     ]
1091     if len(selected) == 1:
```

```
        return selected[0]
    elif len(selected) > 1:
        return "Two or more uses/Possible Error from 'Confidential' Stamp"
    else:
        return "Other"

df["planned_use_former_use"] = df.apply(determine_planned_use, axis=1)

type_of_work_columns = {
    "new_well": "New Well",
    "keyword_destruction_well": "Well Destruction",
    "deepen_well": "Deepen Well",
    "replacement": "Replacement Well",
    "recondition_other.": "Well Reconditioning/Other"
}

def determine_type_of_work(row):
    selected = [
        label for col, label in type_of_work_columns.items()
        if row.get(col, "").strip() == ":selected:"
    ]
    if len(selected) == 1:
        return selected[0]
    elif len(selected) > 1:
        return "Two or more types"
    else:
        return "Unknown"

df["Type of Work"] = df.apply(determine_type_of_work, axis=1)

def map_record_type(type_of_work):
    if type_of_work == "New Well":
        return "WellCompletion/New/Production or Monitoring/NA"
    elif type_of_work == "Unknown":
        return "Unknown"
    elif type_of_work in ["Drill and Destroy", "Destroy"]:
        return "WellCompletion/Drill and Destroy/NA/NA"
    elif type_of_work == "Well Destruction":
        return "WellCompletion/Destruction/NA/NA"
    elif type_of_work in ["Deepen Well", "Replacement Well", "Well Reconditioning/Other"]:
        return "WellCompletion/Modification or Repair/Production or Monitoring/NA"
    else:
        return "Unknown"

df["Record Type"] = df["Type of Work"].apply(map_record_type)

df["final_apn"] = final_apn
df["final_address"] = final_address
df["match_source"] = match_source
df["fuzzy_candidates_log"] = fuzzy_logs
```

```
1142
1143 df.loc[df["template_id"] == "Template 8", "planned_use_former_use"] =
     df.loc[df["template_id"] == "Template 8", "Planned Use"]
1144 df.loc[df["template_id"] == "Template 8", "Type of Work"] = df.loc[df["template_id"] ==
     "Template 8", "Activity"]
1145 df["Record Type"] = df["Type of Work"].apply(map_record_type)
1146
1147 df.to_excel(output_excel, index=False)
1148 print(f"Cleaned data written to: {output_excel}")
```

C:\Well Layer Update\Python\Azure\Data Matching and Merging Script.py

```python
import pandas as pd
import re
from rapidfuzz import process, fuzz
import logging
import numpy as np
from collections import defaultdict

BASE_DIR = r"<internal_project_directory>"

# === File paths ===
file1_path = f"{BASE_DIR}\OCR(SCC1).xlsx"
file2_path = f"{BASE_DIR}\DWR1.xlsx"
file_monterey = f"{BASE_DIR}\Well Completion Reports Monterey.xlsx"
matched_output = f"{BASE_DIR}\matched_output.xlsx"
unmatched_output = f"{BASE_DIR}\unmatched_output.xlsx"
unmatched_ocr_output = f"{BASE_DIR}\unmatched_ocr_output.xlsx"
unmatched_dwr_output = f"{BASE_DIR}\unmatched_dwr_output.xlsx"
supplemented_output_path = f"{BASE_DIR}\matched_output_supplemented.xlsx"
parcel_excel = f"{BASE_DIR}\Parcels.xlsx"




#Load input datasets (OCR, DWR, Monterey, and parcel data) and preserve original match_source
df1 = pd.read_excel(file1_path)
df2 = pd.read_excel(file2_path)
df3 = pd.read_excel(file_monterey)
parcels_df = pd.read_excel(parcel_excel)

if 'match_source' in df1.columns:
    df1['ocr_original_match_source'] = df1['match_source']

# Define helper function to clean and normalize well IDs
def clean_id(raw):
    if pd.isna(raw):
        return ""
    original = str(raw).strip()
    cleaned = original.replace(" ", "").upper().strip()
    # Convert 'c' to 'E' for IDs starting with 'c' or 'C'
    if cleaned.lower().startswith('c'):
        cleaned = 'E' + cleaned[1:]
    if re.match(r"^WCR\d{4}-\d{6}$", cleaned):
        return cleaned
    text = original.lower().replace(",", "")
    text = re.sub(r"no\.?", "", text)
    text = text.replace("/", " ")
    text = re.sub(r"[\s\n]+", " ", text)
```

```
    match = re.match(r"^(\d{1,2})[\.-](\d{1,2})[\.-](\d{2,4})(\d{3,8})$", text)
    if match:
        year_str = match.group(3)
        try:
            year = int(year_str)
            if year < 100:
                year += 1900 if year >= 35 else 2000
            if 1935 <= year <= 2025:
                return match.group(4)
        except ValueError:
            pass
    prefix_removed = re.sub(r"^\d{1,2}[\.-]\d{1,2}[\.-]\d{2,4}", "", text).strip()
    segments = re.split(r"[\s]+", prefix_removed)
    for segment in reversed(segments):
        segment_digits = re.findall(r"\d{3,8}", segment)
        if segment_digits:
            return segment_digits[-1].lstrip('0')
    digits = re.findall(r"\d{3,8}", text)
    if digits:
        return digits[-1].lstrip('0')
    # Return cleaned ID if it starts with 'E' and has digits
    if cleaned.startswith('E') and any(c.isdigit() for c in cleaned[1:]):
        return cleaned
    return ""

#Format APN strings to xxx-yyy-zz pattern for 8-digit APNs, consistent with SCC Parcel db
def format_apn_with_dashes(apn_str):
    apn_str = str(apn_str).strip()
    clean_apn = re.sub(r'\D', '', apn_str)
    if len(clean_apn) == 8:
        return f"{clean_apn[:3]}-{clean_apn[3:6]}-{clean_apn[6:]}"
    else:
        return apn_str

# Normalize parcel APNs by removing non-digits and zero-padding to 8 digits
def normalize_parcel_apn(apn_str):
    return re.sub(r"\D", "", str(apn_str)).zfill(8)

# Generate APN signature by keeping non-zero digits
def apn_signature(apn):
    return ''.join([ch for ch in str(apn) if ch.isdigit() and ch != '0'])

#Clean address strings for consistent matching
def clean_for_matching(addr):
    if not isinstance(addr, str):
        addr = str(addr) if not pd.isna(addr) else ""
    addr = addr.upper()
    addr = addr.replace("(", "").replace(")", "")
    addr = re.sub(r"\bARE\b", "AVE", addr)
    addr = re.sub(r"\bSTH\b", "ST", addr)
```

```
    addr = re.sub(r"\bDRI\b", "DR", addr)
    addr = re.sub(r"\bMOUNT\b", "MT", addr)
    addr = re.sub(r"\bR\b", "RD", addr)
    addr = re.sub(r" STREET", " ST", addr)
    addr = re.sub(r" AVENUE", " AVE", addr)
    addr = re.sub(r" ROAD", " RD", addr)
    addr = re.sub(r" DRIVE", " DR", addr)
    addr = re.sub(r" LANE", " LN", addr)
    addr = re.sub(r" COURT", " CT", addr)
    addr = re.sub(r" PLACE", " PL", addr)
    addr = re.sub(r" TERRACE", " TER", addr)
    addr = re.sub(r" CIRCLE", " CIR", addr)
    addr = re.sub(r" TRAIL", " TRL", addr)
    addr = re.sub(r" CREEK", " CRK", addr)
    addr = re.sub(r" WAY", " WAY", addr)
    addr = re.sub(r"\s+[A-Z\s]*\sCA\s\d{5}$", "", addr)
    addr = re.sub(r"[^\w\s]+$", "", addr)
    addr = re.sub(r"\s+", " ", addr).strip()
    addr = re.sub(r"\b\w\b$", "", addr).strip()
    return addr

# Normalize APN strings by removing non-digits and handling specific cases
def normalize_apn(apn_str):
    digits = re.sub(r"\D", "", str(apn_str))
    if len(digits) == 9 and digits[-3] == '0':
        digits = digits[:-3] + digits[-2:]
    return digits.zfill(8)

# Generate APN signature for matching by keeping non-zero digits
def get_apn_signature(apn_str):
    return "".join(d for d in str(apn_str) if d.isdigit() and d != "0")

# Extract APN from location text using regex
def extract_apn_from_location(location_str):
    match = re.search(r'\bAPN\s*(\d{2,3}[- ]?\d{2,3}[- ]?\d{2,3})\b', str(location_str),
re.IGNORECASE)
    if match:
        return normalize_apn(match.group(1))
    return None

#Normalize year values for comparisons
def normalize_year(v):
    if pd.isna(v) or not str(v).strip():
        return None
    s = str(v).strip()
    if '/' in s:
        parts = s.split('/')
        if len(parts) >= 3 and parts[2].isdigit():
            y = int(parts[2])
            if len(parts[2]) == 2:
```

```
                y += 1900 if y >= 35 else 2000
            return y
    s = re.sub(r'\s', '', s)
    match = re.search(r'(\d{2,4})$', s)
    if match:
        y = int(match.group(1))
        if 0 <= y <= 99:
            y += 1900 if y >= 35 else 2000
        if 1935 <= y <= 2030:
            return y
    return None

# Find best matching DWR record based on well ID and field comparisons
def get_best_match(well_id, records, df1_row=None, fields_to_check=None, field_map=None,
used_indices=None, target_df=None):
    if used_indices is None:
        used_indices = set()

    if not well_id.strip():
        return None, None, None, None, None, 0

    well_id_clean = str(well_id).strip().rstrip('M').replace('.0', '')
    if well_id_clean.startswith('E'):
        well_id_normalized = 'E' + well_id_clean[1:].lstrip('0')
    else:
        well_id_normalized = well_id_clean.lstrip('0')

    exact_matches = []
    for idx, (target_id, wcr_number, apn, addr, depth, water) in enumerate(records):
        if idx in used_indices:
            continue
        target_clean = str(target_id).strip().rstrip('M').replace('.0', '')
        if target_clean.startswith('E'):
            target_normalized = 'E' + target_clean[1:].lstrip('0')
        else:
            target_normalized = target_clean.lstrip('0')

        if well_id_normalized == target_normalized:
            exact_matches.append((target_id, wcr_number, apn, addr, depth, water, idx))

    if exact_matches:
        if len(exact_matches) == 1:
            target_id, wcr_number, apn, addr, depth, water, idx = exact_matches[0]
            logging.debug(f"Exact match: well_id_normalized='{well_id_normalized}',
target='{target_id}' (single match)")
            return target_id, 100, apn, addr, idx, 0
        else:
            logging.debug(f"Multiple matches found for
well_id_normalized='{well_id_normalized}': {len(exact_matches)} candidates")
            best_match = None
            best_match_score = -1
```

```
            best_match_idx = None
            best_matched_fields = []

            for match in exact_matches:
                target_id, wcr_number, apn, addr, depth, water, idx = match
                df2_row = df2.iloc[idx] if target_df is None else target_df.iloc[idx]
                match_score = 0
                matched_fields = []

                for df1_field in ["date", "depth_well_cleaned"] + [f for f in fields_to_check if f not in ["date", "depth_well_cleaned"]]:
                    df2_field = None
                    for key, (df1_col, _) in field_map.items():
                        if df1_col == df1_field:
                            df2_field = key
                            break
                    if not df2_field:
                        continue

                    df1_value = df1_row.get(df1_field) if df1_row is not None else np.nan
                    df2_value = df2_row.get(df2_field)

                    if isinstance(df1_value, str) and df1_value.strip() == '':
                        df1_value = np.nan
                    if isinstance(df2_value, str) and df2_value.strip() == '':
                        df2_value = np.nan

                    if df1_field == "date":
                        y1 = normalize_year(df1_value)
                        y2 = normalize_year(df2_value)
                        if y1 == y2 and y1 is not None:
                            match_score += 2
                            matched_fields.append(df1_field)
                            continue
                    elif df1_field == "depth_well_cleaned":
                        if not pd.isna(df1_value) and not pd.isna(df2_value):
                            if abs(df1_value - df2_value) <= 5:
                                match_score += 2
                                matched_fields.append(df1_field)
                            continue
                    elif df1_field in ["water_static_cleaned", "total_depth_cleaned", "yield_cleaned"]:
                        if not pd.isna(df1_value) and not pd.isna(df2_value):
                            if abs(df1_value - df2_value) <= 5:
                                match_score += 1
                                matched_fields.append(df1_field)
                            continue
                    elif df1_field == "apn_cleaned":
                        v1 = normalize_apn(df1_value) if df1_value else ''
                        v2 = normalize_apn(df2_value) if df2_value else ''
```

```
                        if v1 == v2 and v1:
                            match_score += 1
                            matched_fields.append(df1_field)
                            continue

                    if pd.isna(df1_value) and pd.isna(df2_value):
                        continue
                    elif isinstance(df1_value, str) and isinstance(df2_value, str):
                        if df1_value.strip().upper() == df2_value.strip().upper():
                            match_score += 1
                            matched_fields.append(df1_field)
                    elif df1_value == df2_value:
                        match_score += 1
                        matched_fields.append(df1_field)

                if match_score > best_match_score:
                    best_match = match
                    best_match_score = match_score
                    best_match_idx = idx
                    best_matched_fields = matched_fields

            if best_match:
                target_id, wcr_number, apn, addr, depth, water, idx = best_match
                logging.debug(f"Selected match for well_id_normalized='{well_id_normalized}' based on field matching (score={best_match_score}, matched_fields={best_matched_fields})")
                return target_id, 100, apn, addr, idx, best_match_score

            logging.debug(f"no match from field scoring for well_id_normalized='{well_id_normalized}' - falling back to depth/water")
            best_match = exact_matches[0]
            best_depth_match = False
            df1_depth = df1_row.get("depth_well_cleaned", np.nan) if df1_row is not None else np.nan
            df1_water = df1_row.get("water_static_cleaned", np.nan) if df1_row is not None else np.nan
            for match in exact_matches:
                target_id, wcr_number, apn, addr, depth, water, idx = match
                depth_match = (not pd.isna(df1_depth) and not pd.isna(depth) and abs(df1_depth - depth) <= 5)
                water_match = (not pd.isna(df1_water) and not pd.isna(water) and abs(df1_water - water) <= 5)
                if depth_match:
                    logging.debug(f"Selected match for well_id_normalized='{well_id_normalized}' due to depth match (df1_depth={df1_depth}, candidate_depth={depth})")
                    return target_id, 100, apn, addr, idx, 0
                elif water_match and not best_depth_match:
                    best_match = match
                    logging.debug(f"Selected match for well_id_normalized='{well_id_normalized}' due to water match (df1_water={df1_water}, candidate_water={water})")
```

```
                    best_depth_match = False
            target_id, wcr_number, apn, addr, depth, water, idx = best_match
            logging.debug(f"Fallback match: well_id_normalized='{well_id_normalized}', target='{target_id}' (index={idx})")
            return target_id, 100, apn, addr, idx, 0

    df1_depth = df1_row.get("depth_well_cleaned", np.nan) if df1_row is not None else np.nan
    df1_water = df1_row.get("water_static_cleaned", np.nan) if df1_row is not None else np.nan
    for idx, (target_id, wcr_number, apn, addr, depth, water) in enumerate(records):
        if idx in used_indices:
            continue
        target_clean = str(target_id).strip().rstrip('M').replace('.0', '')
        if target_clean.startswith('E'):
            target_normalized = 'E' + target_clean[1:].lstrip('0')
        else:
            target_normalized = target_clean.lstrip('0')

        is_substring_match = False
        if well_id_normalized in target_normalized and len(target_normalized) > len(well_id_normalized):
            is_substring_match = True
            match_type = f"Substring match (well_id in target): well_id_normalized='{well_id_normalized}', target='{target_clean}'"
        elif target_normalized in well_id_normalized and len(well_id_normalized) > len(target_normalized):
            is_substring_match = True
            match_type = f"Substring match (target in well_id): well_id_normalized='{well_id_normalized}', target='{target_clean}'"

        if is_substring_match:
            depth_match = (not pd.isna(df1_depth) and not pd.isna(depth) and abs(df1_depth - depth) <= 5)
            water_match = (not pd.isna(df1_water) and not pd.isna(water) and abs(df1_water - water) <= 5)
            if not (depth_match or water_match):
                logging.debug(f"Skipping substring match for well_id_normalized='{well_id_normalized}', target='{target_clean}' due to no depth/water match")
                continue

            logging.debug(match_type)
            return target_id, 80, apn, addr, idx, 0

    logging.debug(f"No match found for well_id_normalized='{well_id_normalized}'")
    return None, None, None, None, None, 0

# Perform fuzzy matching on addresses against parcel data
def fuzzy_address_lookup(addr):
    cleaned_input = clean_for_matching(addr)
    if not cleaned_input:
```

```
        return None, None, 0

    matches = process.extract(
        cleaned_input,
        parcels_df["parcel_address"].tolist(),
        scorer=fuzz.token_sort_ratio,
        limit=10
    )

    for match_text, score, idx in matches:
        if score >= 85:
            best_row = parcels_df.iloc[idx]
            return best_row["parcel_apn"], best_row["parcel_address"], score

    return None, None, 0

# Prepare DWR dataset (df2) for matching by cleaning IDs, APNs, and addresses
df2["clean_apn"] = df2.iloc[:, 27].apply(normalize_apn)
df2["clean_address"] = df2.iloc[:, 3].apply(clean_for_matching)
df2_depth = df2["Total Completed Depth"].astype(float).fillna(np.nan)
df2_water = df2["Static Water Level"].astype(float).fillna(np.nan)

df2_ids = []
for i in range(len(df2)):
    id1_raw = df2.iloc[i, 1]  # Legacy Log Number
    id0_raw = df2.iloc[i, 0]  # WCR Number
    id1 = clean_id(id1_raw)
    id0 = clean_id(id0_raw)
    if id1 and re.match(r"^(WCR\d{4}-\d{6}|\d{3,8}|E\d+)$", id1):
        df2_ids.append(id1)
    else:
        df2_ids.append(id0)
df2_ids = pd.Series(df2_ids, index=df2.index)

df2_apn = df2.iloc[:, 27].astype(str).fillna('')
df2_addr = df2.iloc[:, 3].astype(str).fillna('')
df2_wcr = df2.iloc[:, 0].astype(str).fillna('')
df2_records = list(zip(df2_ids, df2_wcr, df2_apn, df2_addr, df2_depth, df2_water))

# Prepare Monterey dataset (df3) for secondary matching
df3.columns = df3.columns.str.strip()
df3_ids = []
for i in range(len(df3)):
    id1_raw = df3.iloc[i, 1]  # Legacy Log Number
    id0_raw = df3.iloc[i, 0]  # WCR Number
    id1 = clean_id(id1_raw)
    id0 = clean_id(id0_raw)
    if id1 and re.match(r"^(WCR\d{4}-\d{6}|\d{3,8}|E\d+)$", id1):
        df3_ids.append(id1)
    else:
```

```
        df3_ids.append(id0)
df3_ids = pd.Series(df3_ids, index=df3.index)
df3_wcr = df3.iloc[:, 0].astype(str).fillna('')
df3_apn = df3.iloc[:, 27].astype(str).fillna('')
df3_addr = df3.iloc[:, 3].astype(str).fillna('')
df3_depth = df3["Total Completed Depth"].astype(float).fillna(np.nan)
df3_water = df3["Static Water Level"].astype(float).fillna(np.nan)
df3_records = list(zip(df3_ids, df3_wcr, df3_apn, df3_addr, df3_depth, df3_water))

# Prepare parcel data for APN and address lookups
parcels_df["parcel_address"] = parcels_df.iloc[:, 13].astype(str).apply(clean_for_matching)
parcels_df["parcel_apn"] = parcels_df.iloc[:, 1].apply(normalize_parcel_apn)
parcel_apn_to_address = dict(zip(parcels_df["parcel_apn"], parcels_df["parcel_address"]))
signature_to_apns = defaultdict(list)
for apn in parcels_df["parcel_apn"]:
    sig = get_apn_signature(apn)
    signature_to_apns[sig].append(apn)
apn_lookup = parcels_df.set_index("parcel_apn")["parcel_address"].to_dict()

# Set up logging for debugging and tracking matches
logging.basicConfig(
    filename=r"C:\Well Layer Update\Python\Azure\Results\wc11\excel 3\fuzzy_match_debug.log",
    filemode="a",
    format="%(asctime)s [%(levelname)s] %(message)s",
    level=logging.DEBUG
)

# Log sample data for debugging
logging.debug(f"df1 well_id_cleaned sample: {df1['well_id_cleaned'].head(10).tolist()}")
logging.debug(f"df2_ids sample: {df2_ids.head(10).tolist()}")


# Define fields for validating duplicate matches
fields_to_check = [
    "total_depth_cleaned",
    "depth_well_cleaned",
    "water_static_cleaned",
    "yield_cleaned",
    "permit_date",
    "permit_no",
    "date",
    "apn_cleaned"
]

#Define mapping of DWR fields to OCR fields and their source columns
field_map = {
    "Owner Assigned Well Number": ("owner_well_id", "owner_assigned_source"),
    "Date Work Ended": ("date", "date_source"),
    "Permit Number": ("permit_no", "permit_number_source"),
    "Driller Name": ("contractor", "driller_name_source"),
```

```
    "Total Drill Depth": ("total_depth_cleaned", "total_drill_depth_source"),
    "Total Completed Depth": ("depth_well_cleaned", "total_completed_depth_source"),
    "Static Water Level": ("water_static_cleaned", "static_water_level_source"),
    "Well Yield": ("yield_cleaned", "well_yield_source"),
    "B118WellUse": ("planned_use_former_use", "b118welluse_source"),
    "Record Type": ("Type of Work", "record_type_source"),
    "Permit Date": ("permit_date", "permit_date_source"),
    "Driller License Number": ("contractor_license", "driller_license_number_source"),
    "Drilling Method": ("drilling_method", "drilling_method_source"),
    "Fluid": ("fluid", "fluid_source"),
    "Total Draw Down": ("total_drawdown", "total_drawdown_source"),
    "Test Type": ("test_type", "test_type_source"),
    "Pump Test Length": ("pump_test_length_cleaned", "pump_test_length_source"),
    "Other Observations": ("other_observations", "other_observations_source"),
    "City": ("city", "city_source"),
    "Ground Surface Elevation": ("surface_elevation", "surface_elevation_source"),
    "Elevation Determination Method": ("elevation_method", "elevation_method_source"),
    "Top Of Perforated Interval": ("top_perforations", "top_perforations_source"),
    "Bottom of Perforated Interval": ("bottom_perforations", "bottom_perforations_source"),
    "Decimal Latitude": ("lat_cleaned", "latitude_source"),
    "Decimal Longitude": ("long_cleaned", "longitude_source"),
    "APN": ("apn_cleaned", "apn_source")
}

#Define additional OCR fields to carry over to supplemented output
additional_fields = {
    "seal_cleaned": ("seal_cleaned", "seal_cleaned_source"),
    "first_water_cleaned": ("first_water_cleaned", "first_water_cleaned_source"),
    "template_id": ("template_id", "template_id_source"),
    "OCR_match_source": ("ocr_original_match_source", "OCR_match_source_source")
}

#Initialize set to track used DWR indices for matching
used_dwr_indices = set()

# Match OCR records to DWR records and update APN/address
def match_apn_and_address(row):
    well_id = clean_id(row["well_id_cleaned"]).strip()
    if not well_id:
        return row["final_address"], str(row["final_apn"]).zfill(7), "No match", "No valid
well ID", 0

    well_id = str(well_id).strip().lstrip('0').replace('.0', '')
    logging.debug(f"Processing well_id: {well_id}")

    match_id, score, apn_dwr, addr_dwr, dwr_idx, match_score = get_best_match(
        well_id, df2_records, df1_row=row, fields_to_check=fields_to_check,
field_map=field_map, used_indices=used_dwr_indices, target_df=df2
    )
    if match_id and dwr_idx is not None:
```

```
        dwr_row = df2.iloc[dwr_idx]
        wcr_number = dwr_row["WCR Number"]
        dwr_apn = dwr_row["clean_apn"]
        dwr_address = dwr_row["clean_address"]
        logging.debug(f"Match found for well_id='{well_id}' to df2_id='{match_id}' (score={score}, duplicate_match_score={match_score})")
        if not dwr_apn or dwr_apn == '00000000':
            potential_apn = extract_apn_from_location(dwr_row["Well Location"])
            if potential_apn:
                dwr_apn = potential_apn
                logging.debug(f"Extracted APN '{dwr_apn}' from Well Location")

        if dwr_apn in parcel_apn_to_address:
            return parcel_apn_to_address[dwr_apn], dwr_apn, f"ID match (score {score}) → APN match", f"OCR ID '{well_id}' → DWR ID '{match_id}' | WCR Number: {wcr_number}", match_score

        matched_apn, matched_address, addr_score = fuzzy_address_lookup(dwr_address)
        if matched_apn and matched_address:
            return (
                matched_address,
                matched_apn,
                f"ID match (score {score}) → fuzzy address",
                f"OCR ID '{well_id}' → DWR ID '{match_id}' → Address (score: {addr_score}) | WCR Number: {wcr_number}",
                match_score
            )

        logging.debug(f"ID match for '{well_id}' to '{match_id}' (score: {score}) but no parcel match")
        return row["final_address"], str(row["final_apn"]).zfill(7), f"ID match (score {score}) with no parcel match", f"OCR ID '{well_id}' → DWR ID '{match_id}' | WCR Number: {wcr_number}", match_score

    location_details_raw = row.get("location_details", "")
    location_details_clean = clean_for_matching(location_details_raw)

    if location_details_clean:
        dwr_locations_raw = df2["Well Location"].fillna("").astype(str)
        dwr_locations_cleaned = dwr_locations_raw.apply(clean_for_matching).tolist()

        match_result = process.extractOne(location_details_clean, dwr_locations_cleaned, scorer=fuzz.token_sort_ratio)

        if match_result:
            best_match_text, score, match_index = match_result
            matched_row = df2.iloc[match_index]
            original_dwr_location = dwr_locations_raw.iloc[match_index]

            if score >= 80:
                legacy_log = str(matched_row.get("Legacy Log Number", "")).strip().upper()
                if legacy_log != "NN":
```

```
                    ocr_apn = str(row["final_apn"]).zfill(7)
                    ocr_match_source = str(row.get("match_source", ""))
                    if ocr_match_source == "No match" or ocr_apn not in parcel_apn_to_addres-
s:
                        ocr_apn = "0000000"
                    return row["final_address"], ocr_apn, "Fallback to OCR values", "Fuzzy
match blocked: Legacy Log Number not 'NN'", 0

                dwr_apn = matched_row["clean_apn"]
                dwr_address = matched_row["clean_address"]

                if not dwr_apn or dwr_apn == '00000000':
                    potential_apn = extract_apn_from_location(matched_row["Well Location"])
                    if potential_apn:
                        dwr_apn = potential_apn

                if dwr_apn in parcel_apn_to_address:
                    wcr_number = matched_row["WCR Number"]
                    return parcel_apn_to_address[dwr_apn], dwr_apn, "Fuzzy location → APN
match", f"Location '{location_details_raw}' → '{original_dwr_location}' | WCR Number:
{wcr_number}", 0

                matched_apn, matched_address, addr_score = fuzzy_address_lookup(dwr_address)
                if matched_apn and matched_address:
                    wcr_number = matched_row["WCR Number"]
                    return (
                        matched_address,
                        matched_apn,
                        "Fuzzy location → fuzzy address",
                        f"Location '{location_details_raw}' → '{original_dwr_location}' →
Address: {dwr_address} (score: {score}) → Parcel (score: {addr_score}) | WCR Number:
{wcr_number}",
                        0
                    )

                wcr_number = matched_row["WCR Number"]
                return row["final_address"], str(row["final_apn"]).zfill(7), "Fuzzy location
match with no parcel match", f"Location '{location_details_raw}' → '{original_dwr_location}'
(score: {score}) | WCR Number: {wcr_number}", 0

    ocr_apn = str(row["final_apn"]).zfill(7)
    ocr_match_source = str(row.get("match_source", ""))
    if ocr_match_source == "No match":
        ocr_apn = "0000000"
    return row["final_address"], ocr_apn, "Fallback to OCR values", "No match found", 0

# Apply matching logic to update APN and address fields in OCR dataset
df1_ids = df1["well_id_cleaned"].apply(clean_id)
df1_apn = df1["final_apn"].astype(str).fillna('')
df1_addr = df1["final_address"].astype(str).fillna('')

```

```
df1[["final_address", "final_apn", "match_source", "match_log", "duplicate_match_score"]] = df1.apply(
    lambda row: pd.Series(match_apn_and_address(row)),
    axis=1
)

#  Process matched and unmatched records, including Monterey dataset checks
matched_records = []
matched_dwr_ids = set()
unmatched_ocr_records = []

for i in range(len(df1)):
    source_id = df1_ids.iloc[i]
    match_source = df1.at[i, "match_source"]

    if match_source == "Fallback to OCR values":
        matched_in_monterey = "No"
        match_id_mon, score_mon, apn_mon, addr_mon, mon_idx, _ = get_best_match(source_id, df3_records, df1_row=df1.iloc[i], fields_to_check=fields_to_check, field_map=field_map, target_df=df3)
        print(f"NO MATCH: df1_id raw='{df1.at[i, 'well_id']}', cleaned='{source_id}'")
        if match_id_mon:
            matched_in_monterey = "Yes"
        unmatched_record = df1.iloc[i].copy()
        unmatched_record["Matched_in_Monterey"] = matched_in_monterey
        unmatched_ocr_records.append(unmatched_record)

    else:
        df1_depth = df1.at[i, "depth_well_cleaned"] if "depth_well_cleaned" in df1.columns else np.nan
        df1_water = df1.at[i, "water_static_cleaned"] if "water_static_cleaned" in df1.columns else np.nan
        match_id, score, apn_dwr, addr_dwr, dwr_index, match_score = get_best_match(source_id, df2_records, df1_row=df1.iloc[i], fields_to_check=fields_to_check, field_map=field_map, used_indices=used_dwr_indices, target_df=df2)
        if match_id and dwr_index is not None and dwr_index not in used_dwr_indices:
            logging.debug(f"[Row {i}] Match found for df1_id='{source_id}' to df2_id='{match_id}' (score={score}, duplicate_match_score={match_score})")
            record = df2.iloc[dwr_index].copy()
            record["final_address"] = df1.at[i, "final_address"]
            record["final_apn"] = df1.at[i, "final_apn"]
            record["match_source"] = match_source if match_source else f"ID match (score {score})"
            record["matched_df2_well_id"] = match_id
            record["matched_df1_well_id"] = source_id
            record["duplicate_match_score"] = match_score
            record["df1_row_index"] = i
            matched_records.append(record)
            matched_dwr_ids.add(match_id)
            used_dwr_indices.add(dwr_index)
        else:
```

```
            logging.debug(f"[Row {i}] No df2 match for df1_id='{source_id}' or df2 index {dwr_index} already used")
            unmatched_record = df1.iloc[i].copy()
            unmatched_record["Matched_in_Monterey"] = "No"
            unmatched_ocr_records.append(unmatched_record)

    if match_source in [
        "Fuzzy location → APN match",
        "Fuzzy location → fuzzy address",
        "Fuzzy location match with no parcel match"
    ]:
        location_raw = df1.at[i, "location_details"]
        location_clean = clean_for_matching(location_raw)
        dwr_locations_raw = df2["Well Location"].fillna("").astype(str)
        dwr_locations_clean = dwr_locations_raw.apply(clean_for_matching).tolist()

        match_result = process.extractOne(
            location_clean,
            dwr_locations_clean,
            scorer=fuzz.token_sort_ratio
        )

        if match_result:
            match_value, score, match_index = match_result
            if match_index not in used_dwr_indices:
                logging.debug(f"[Row {i}] Best fuzzy match='{match_value}' with score={score}")
                matched_row = df2.iloc[match_index].copy()
                matched_row["final_address"] = df1.at[i, "final_address"]
                matched_row["final_apn"] = df1.at[i, "final_apn"]
                matched_row["match_source"] = match_source
                matched_row["matched_df2_well_id"] = df2_ids.iloc[match_index]
                matched_row["matched_df1_well_id"] = source_id
                matched_row["duplicate_match_score"] = 0
                matched_row["df1_row_index"] = i
                matched_records.append(matched_row)
                matched_dwr_ids.add(df2_ids.iloc[match_index])
                used_dwr_indices.add(match_index)
            else:
                logging.debug(f"[Row {i}] Fuzzy match index {match_index} already used")
                unmatched_record = df1.iloc[i].copy()
                unmatched_record["Matched_in_Monterey"] = "No"
                unmatched_ocr_records.append(unmatched_record)
        else:
            logging.warning(f"[Row {i}] Fuzzy match expected but no location match found for location='{location_clean}'")
            unmatched_record = df1.iloc[i].copy()
            unmatched_record["Matched_in_Monterey"] = "No"
            unmatched_ocr_records.append(unmatched_record)

# Generate unmatched DWR records (those not matched to OCR)
```

```
651 unmatched_dwr_records = df2[~df2_ids.isin(matched_dwr_ids)]
652
653 # Save matched and unmatched records to Excel files
654 if matched_records:
655     matched_df = pd.DataFrame(matched_records)
656     matched_df["final_apn"] = matched_df["final_apn"].fillna("").astype(str)
657     mask = (matched_df["match_source"] == "No match") & (
658         matched_df["final_address"].isna() | (matched_df["final_address"].str.strip() == "")
659     )
660     matched_df.loc[mask, "final_apn"] = "0000000"
661
662     fallback_lookup = df1.set_index("well_id_cleaned")["match_source"].to_dict()
663     matched_df["fallback_match_source"] = ""
664
665     for idx, row in matched_df.iterrows():
666         if row.get("match_source") == "ID match with no parcel match":
667             well_id = row.get("matched_df1_well_id")
668             if well_id in fallback_lookup:
669                 matched_df.at[idx, "fallback_match_source"] = fallback_lookup[well_id]
670
671     matched_df.to_excel(matched_output, index=False)
672     print(f"Matched DWR records written to: {matched_output}")
673
674 if unmatched_ocr_records:
675     unmatched_ocr_df = pd.DataFrame(unmatched_ocr_records)
676     unmatched_ocr_df.to_excel(unmatched_ocr_output, index=False)
677     print(f"Unmatched OCR records written to: {unmatched_ocr_output}")
678
679 if not unmatched_dwr_records.empty:
680     unmatched_dwr_records.to_excel(unmatched_dwr_output, index=False)
681     print(f"Unmatched DWR records written to: {unmatched_dwr_output}")
682
683 # Format APN fields in matched and unmatched OCR datasets
684 matched_df = pd.DataFrame(matched_records)
685 unmatched_ocr_df = pd.DataFrame(unmatched_ocr_records)
686
687 if not matched_df.empty:
688     matched_df["final_apn"] = matched_df["final_apn"].apply(format_apn_with_dashes)
689
690 if not unmatched_ocr_df.empty:
691     unmatched_ocr_df["final_apn"] = unmatched_ocr_df["final_apn"].apply(format_apn_with_dash-
    es)
692
693 # Supplement matched dataset with OCR fields and source tracking
694 supplemented_df = matched_df.copy().reset_index(drop=True)
695
696 supplemented_df["duplicate_exists"] = ""
697 for _, source_col in list(field_map.values()) + list(additional_fields.values()):
698     supplemented_df[source_col] = ""
699 for final_col, _ in additional_fields.items():
```

```
    supplemented_df[final_col] = np.nan

# create column for the OCR depths flag
supplemented_df["depths_flag"] = ""

columns_to_supplement = [
    (final_col, ocr_col, source_col)
    for final_col, (ocr_col, source_col) in field_map.items()
]
additional_columns = [
    (final_col, ocr_col, source_col)
    for final_col, (ocr_col, source_col) in additional_fields.items()
]

df1["well_id_cleaned"] = df1["well_id_cleaned"].apply(lambda x: str(x).strip().rstrip(".0") if not pd.isna(x) else "")

for idx, row in supplemented_df.iterrows():
    df1_idx = row.get("df1_row_index")
    if pd.isna(df1_idx):
        continue
    ocr_row = df1.iloc[int(df1_idx)]
    supplemented_df.at[idx, "duplicate_exists"] = ocr_row.get("duplicate_exists", "")
    for match_col, ocr_col, source_col in columns_to_supplement:
        original_val = supplemented_df.at[idx, match_col]
        ocr_val = ocr_row.get(ocr_col, np.nan)
        if match_col == "B118WellUse":
            dwr_val_str = str(original_val).strip().upper() if isinstance(original_val, str) else ""
            ocr_val_str = str(ocr_val).strip()
            if dwr_val_str == "UNKNOWN" and ocr_val_str.upper() not in {"", "NAN"}:
                supplemented_df.at[idx, match_col] = ocr_val
                supplemented_df.at[idx, source_col] = "AI"
                continue
        if pd.isna(original_val) or (isinstance(original_val, str) and original_val.strip() == ""):
            if not pd.isna(ocr_val) and str(ocr_val).strip() != "":
                supplemented_df.at[idx, match_col] = ocr_val
                supplemented_df.at[idx, source_col] = "AI"
            else:
                supplemented_df.at[idx, source_col] = "DWR"
        else:
            supplemented_df.at[idx, source_col] = "DWR"
    for match_col, ocr_col, source_col in additional_columns:
        ocr_val = ocr_row.get(ocr_col, np.nan)
        if not pd.isna(ocr_val) and str(ocr_val).strip() != "":
            supplemented_df.at[idx, match_col] = ocr_val
            supplemented_df.at[idx, source_col] = "AI"
        else:
            supplemented_df.at[idx, match_col] = np.nan
            supplemented_df.at[idx, source_col] = "AI"
```

```
748
749
750     # Pull the two source columns that belong to the depth fields
751     completed_src = supplemented_df.at[idx, "total_completed_depth_source"]
752     drill_src     = supplemented_df.at[idx, "total_drill_depth_source"]
753
754     # If either source contains "AI" then copy OCR flag
755     if (pd.notna(completed_src) and "AI" in str(completed_src).upper()) or \
756        (pd.notna(drill_src)     and "AI" in str(drill_src).upper()):
757         ocr_flag = ocr_row.get("depths_flag", "")
758         supplemented_df.at[idx, "depths_flag"] = ocr_flag
759     else:
760         supplemented_df.at[idx, "depths_flag"] = ""   # blank when both are DWR (or empty)
761
762 # Flag record type mismatches between DWR and OCR datasets
763 for idx, row in supplemented_df.iterrows():
764     df1_idx = row.get("df1_row_index")
765     if pd.isna(df1_idx):
766         continue
767     ocr_row = df1.iloc[int(df1_idx)]
768     dwr_record_type = str(row.get("Record Type", "")).strip().upper()
769     ocr_record_type = str(ocr_row.get("Record Type", "")).strip().upper()
770     if ocr_record_type in {"", "UNKNOWN"}:
771         continue
772     if dwr_record_type != ocr_record_type:
773         supplemented_df.at[idx, "Record_Type_Mismatch"] = f"DWR: {dwr_record_type} ≠ OCR:
    {ocr_record_type}"
774
775 #Add fallback match source for records with no parcel match
776 supplemented_df["fallback_match_source"] = ""
777 for idx, row in supplemented_df.iterrows():
778     if row.get("match_source") == "ID match with no parcel match":
779         df1_idx = row.get("df1_row_index")
780         if not pd.isna(df1_idx):
781             fallback_val = df1.iloc[int(df1_idx)].get("match_source", "")
782             supplemented_df.at[idx, "fallback_match_source"] = fallback_val
783
784 # Save supplemented dataset to Excel
785 supplemented_df.to_excel(supplemented_output_path, index=False)
786 print("supplemented file written to:", supplemented_output_path)
```

C:\Well Layer Update\Python\Azure\County Database Matching Script.py

```python
import pandas as pd
import re
import logging



BASE_DIR = r"<internal_project_directory>"
# df4_path = f"{BASE_DIR}\C1.xlsx"
# df5_path = f"{BASE_DIR}\OCR(SCC1)_OCR(DWR1)2.xlsx"
# matched_output = f"{BASE_DIR}\SCC1_DWR1_C1_2.xlsx"
# unmatched_county_output = f"{BASE_DIR}\C2.xlsx"
# unmatched_source_output = f"{BASE_DIR}\OCR(SCC1)_OCR(DWR1)_Final.xlsx"

#c3 and dwr2
df4_path = f"{BASE_DIR}\C3.xlsx"
df5_path = f"{BASE_DIR}\DWR2.xlsx"
matched_output = f"{BASE_DIR}\C3_DWR2.xlsx"
unmatched_county_output = f"{BASE_DIR}\C3.xlsx"
unmatched_source_output = f"{BASE_DIR}\C3_DWR2\DWR3.xlsx"

df4 = pd.read_excel(df4_path)
df5 = pd.read_excel(df5_path)


def clean_id(raw):
    if pd.isna(raw):
        return ""
    original = str(raw).strip()
    # Replace 'O' with '0' and 'e' with 'E'
    cleaned = original.replace("O", "0").replace("o", "0").replace("e", "E").upper().strip()
    if cleaned.lower().startswith('c'):
        cleaned = 'E' + cleaned[1:]
    if re.match(r"^WCR\d{4}-\d{6}$", cleaned):
        return cleaned
    text = original.lower().replace("O", "0").replace("o", "0").replace("e", "E").replace(",", "").replace("no.", "").replace("/", " ")
    text = re.sub(r"[\s\n]+", " ", text)
    match = re.match(r"^(\d{1,2})[\.-](\d{1,2})[\.-](\d{2,4})(\d{3,8})$", text)
    if match:
        year_str = match.group(3)
        try:
            year = int(year_str)
            if year < 100:
                year += 1900 if year >= 35 else 2000
            if 1935 <= year <= 2025:
                return match.group(4)
        except ValueError:
            pass
```

```
    prefix_removed = re.sub(r"^\d{1,2}[\.-]\d{1,2}[\.-]\d{2,4}", "", text).strip()
    segments = re.split(r"[\s]+", prefix_removed)
    for segment in reversed(segments):
        segment_digits = re.findall(r"\d{3,8}", segment)
        if segment_digits:
            return segment_digits[-1].lstrip('0')
    digits = re.findall(r"\d{3,8}", text)
    if digits:
        return digits[-1].lstrip('0')
    if cleaned.startswith('E') and any(c.isdigit() for c in cleaned[1:]):
        return cleaned
    return ""


logging.basicConfig(
    filename=r"C:\Well Layer Update\GIS\master files\county_match_debug.log",
    filemode="a",
    format="%(asctime)s [%(levelname)s] %(message)s",
    level=logging.DEBUG
)


df4_ids = df4["Log_Number"].apply(clean_id)
df5_legacy_ids = df5["Legacy Log Number"].apply(clean_id)  # Apply clean_id for consistency
df5_wcr_ids = df5["WCR Number"].apply(clean_id) if "WCR Number" in df5.columns else
pd.Series([""] * len(df5))


matched_records = []
unmatched_county_records = []
unmatched_source_records = []
used_df5_indices = set()

for i, county_id in enumerate(df4_ids):
    if not county_id.strip():
        continue  # Skip empty Log_Number
    county_id_clean = str(county_id).strip().rstrip('M').replace('.0', '')
    county_id_normalized = 'E' + county_id_clean[1:].lstrip('0') if
county_id_clean.startswith('E') else county_id_clean.lstrip('0')

    # Debug specific case
    if county_id_clean.lower() == 'e0139534' or county_id_clean == 'E0139534':
        logging.debug(f"Debugging e0139534: county_id_clean='{county_id_clean}',
county_id_normalized='{county_id_normalized}'")

    match_found = False
    matched_field = ""
    df5_row = None
    df4_row = df4.iloc[i]

    # try matching with Legacy Log Number
```

```
    for j, legacy_id in enumerate(df5_legacy_ids):
        if j in used_df5_indices:
            continue
        legacy_id_clean = str(legacy_id).strip().rstrip('M').replace('.0', '')
        legacy_id_normalized = 'E' + legacy_id_clean[1:].lstrip('0') if legacy_id_clean.startswith('E') else legacy_id_clean.lstrip('0')


        if county_id_normalized == legacy_id_normalized:
            logging.debug(f"Match found: county_id='{county_id}' to source_legacy_id='{legacy_id}'")
            df5_row = df5.iloc[j].copy()
            matched_field = "Legacy Log Number"
            used_df5_indices.add(j)
            match_found = True
            break

    #if no match on Legacy Log Number, try WCR Number
    if not match_found and "WCR Number" in df5.columns:
        for j, wcr_id in enumerate(df5_wcr_ids):
            if j in used_df5_indices:
                continue
            wcr_id_clean = str(wcr_id).strip().rstrip('M').replace('.0', '')
            wcr_id_normalized = 'E' + wcr_id_clean[1:].lstrip('0') if wcr_id_clean.startswith('E') else wcr_id_clean.lstrip('0')

            if county_id_normalized == wcr_id_normalized:
                logging.debug(f"Match found: county_id='{county_id}' to source_wcr_id='{wcr_id}'")
                df5_row = df5.iloc[j].copy()
                matched_field = "WCR Number"
                used_df5_indices.add(j)
                match_found = True
                break

    if match_found:
        # Append all df4 columns with 'county_' prefix
        df4_row_prefixed = df4_row.rename(lambda x: f"county_{x}")
        # Create record with df5 columns, df4 columns, and match metadata
        record = pd.concat([df5_row, df4_row_prefixed])
        record["matched_field"] = matched_field
        record["cleaned_county_id"] = county_id
        record["source_id"] = legacy_id if matched_field == "Legacy Log Number" else wcr_id
        matched_records.append(record)
    else:
        unmatched_county_records.append(df4_row)


for j in range(len(df5)):
    if j not in used_df5_indices:
        df5_row = df5.iloc[j].copy()
```

```
        # Collect only unmatched df5 records
        unmatched_source_records.append(df5_row)
        logging.debug(f"Unmatched df5 record added: index={j},
source_id='{df5_row.get('Legacy Log Number', 'N/A') if df5_row.get('Legacy Log Number') else
df5_row.get('WCR Number', 'N/A')}'")

# save matched
if matched_records:
    matched_df = pd.DataFrame(matched_records)
    matched_df.to_excel(matched_output, index=False)
    print(f"Matched records written to: {matched_output}")
else:
    print("No matched records to write to matched output.")

# save unmatched
if unmatched_county_records:
    unmatched_county_df = pd.DataFrame(unmatched_county_records)

    if 'IFILE' not in unmatched_county_df.columns:
        unmatched_county_df['IFILE'] = pd.NA
    unmatched_county_df.to_excel(unmatched_county_output, index=False)
    print(f"Unmatched county records written to: {unmatched_county_output}")
else:
    print("No unmatched county records with non-empty Log_Number.")

# save sources
if unmatched_source_records:
    unmatched_source_df = pd.DataFrame(unmatched_source_records)
    unmatched_source_df.to_excel(unmatched_source_output, index=False)
    print(f"Unmatched source records written to: {unmatched_source_output}")
else:
    print("No unmatched source records.")
```

C:\Well Layer Update\Python\Azure\Field Highlighting Script.py

```python
from openpyxl import load_workbook
from openpyxl.styles import Font

BASE_DIR = r"<internal_project_directory>"
supplemented_path = f"{BASE_DIR}\SCC_WELLS_FINAL_table.xlsx"

# Updated field_map to include additional fields
# field_map = {
#     "Owner_Assigned_Well_Number": "owner_assigned_source",
#     "Permit Number": "permit_number_source",
#     "Driller Name": "driller_name_source",
#     "Total Drill Depth": "total_drill_depth_source",
#     "Total Completed Depth": "total_completed_depth_source",
#     "Static Water Level": "static_water_level_source",
#     "Well Yield": "well_yield_source",
#     "B118WellUse": "b118welluse_source",
#     "Record Type": "record_type_source",
#     "Permit Date": "permit_date_source",
#     "Driller License Number": "driller_license_number_source",
#     "Drilling Method": "drilling_method_source",
#     "Fluid": "fluid_source",
#     "Total Draw Down": "total_drawdown_source",
#     "Test Type": "test_type_source",
#     "Pump Test Length": "pump_test_length_source",
#     "Other Observations": "other_observations_source",
#     "City": "city_source",
#     "Ground Surface Elevation": "surface_elevation_source",
#     "Elevation Determination Method": "elevation_method_source",
#     "Top Of Perforated Interval": "top_perforations_source",
#     "Bottom of Perforated Interval": "bottom_perforations_source",
#     "Decimal Latitude": "latitude_source",
#     "Decimal Longitude": "longitude_source",
#     "APN": "apn_source",
#     "seal_cleaned": "seal_cleaned_source",
#     "first_water_cleaned": "first_water_cleaned_source",
#     "template_id": "template_id_source",
#     "OCR_match_source": "OCR_match_source_source"
# }

#after gis export
field_map = {
    "Owner_Assigned_Well_Number": "owner_assigned_source",
    "Permit_Number": "permit_number_source",
    "Date_Work_Ended": "date_source",
    "Driller_Name": "driller_name_source",
    "Total_Drill_Depth": "total_drill_depth_source",
    "Total_Completed_Depth": "total_completed_depth_source",
    "Static_Water_Level": "static_water_level_source",
```

```
    "Well_Yield": "well_yield_source",
    "B118WellUse": "b118welluse_source",
    "Well_Type": "b118welluse_source",
    "Record_Type": "record_type_source",
    "Permit_Date": "permit_date_source",
    "Driller_License_Number": "driller_license_number_source",
    "Drilling_Method": "drilling_method_source",
    "Fluid": "fluid_source",
    "Total_Draw_Down": "total_drawdown_source",
    "Test_Type": "test_type_source",
    "Pump_Test_Length": "pump_test_length_source",
    "City": "city_source",
    "Ground_Surface_Elevation": "surface_elevation_source",
    "Top_Of_Perforated_Interval": "top_perforations_source",
    "Bottom_of_Perforated_Interval": "bottom_perforations_source",
    "APN_mapped": "apn_source",
    "Seal_Depth": "seal_cleaned_source",
    "First_Water": "first_water_cleaned_source"
}


# Load workbook and sheet
wb = load_workbook(supplemented_path)
ws = wb.active


headers = {cell.value: idx for idx, cell in enumerate(ws[1], 1)}  # 1-based indexing

red_font = Font(color="FF0000")

# Get indexes for special fields
final_apn_idx = headers.get("final_apn")
final_address_idx = headers.get("final_address")
match_source_idx = headers.get("match_source")

# Loop through rows starting from row 2 (excluding header)
for row in ws.iter_rows(min_row=2, max_row=ws.max_row):
    # Highlight special logic for final_apn and final_address
    match_source_val = row[match_source_idx - 1].value if match_source_idx else None

    if final_apn_idx:
        apn_cell = row[final_apn_idx - 1]
        if match_source_val not in ["DWR APN matched to parcel database", "Fuzzy DWR address match", "No match"]:
            apn_cell.font = red_font  # Mark as red if not in accepted match sources

    if final_address_idx:
        addr_cell = row[final_address_idx - 1]
        if match_source_val not in ["DWR APN matched to parcel database", "Fuzzy DWR address match"]:
```

```
            addr_cell.font = red_font

    #Apply red font to "AI"-sourced fields
    for field, source_col in field_map.items():
        col_idx = headers.get(field)
        source_idx = headers.get(source_col)

        if not col_idx or not source_idx:
            continue

        source_val = row[source_idx - 1].value
        if source_val == "AI":
            row[col_idx - 1].font = red_font  # Highlight the main field in red

# Save with highlights
wb.save(supplemented_path)
print("Field highlighting complete.")
```