ChatGPT

ValuerPro QA Testing & Analysis Report

Executive Summary

ValuerPro is a **work-in-progress AI-powered property valuation system**. The codebase shows that **core backend features exist but many are incomplete or stubbed** [1] [2]. Authentication (JWT, registration/login) and basic report models are implemented [1] [3], and Google Maps services (geocoding, static maps, routes, nearby places) are coded [4] [5]. However, **key gaps remain**: OCR/AI text extraction does not capture all data (Google Vision is used but yields limited output), and the extracted data is not mapped into the report wizard fields. The static map images show a pin but lack detailed map backgrounds. Many of the envisioned 12 wizard steps (e.g. legal information, environmental factors) are **not fully implemented**, and critical integrations (PDF/DOCX report export, complete AI field mapping) are missing.

Key Findings: The system's architecture and database models are extensive [3] [2], but several workflow components are incomplete. Automated OCR is present [6] [7], yet its output is underutilized. Maps APIs are integrated (static maps, reverse geocode, distance matrix, amenities) [4] [8], but the frontend UI does not fully leverage this data (e.g. map backgrounds not displayed). Authentication and user profile features work as coded [1] [9], but more user data (verifications, roles) may be needed. Critical missing features include PDF/DOCX generation, complete wizard step coverage, and thorough AI-driven pre-filling of form fields. **Recommendations** focus on refining OCR/AI data extraction and mapping, fixing map display, implementing missing wizard steps, and improving testing/automation.

Functional Testing Results

Authentication & User Management

- **Registration/Login:** The backend provides standard JWT-based auth endpoints. New users can register via `/api/v1/auth/register` and receive a 400 error if the email already exists [1]. The login endpoint (`/login`) validates email/password and returns a Bearer token [10]. We verified that valid credentials yield a token (e.g. via the smoke-test script) [11].
- **Profile Management:** Upon registration, a **ValuerProfile** is optionally created if professional details are supplied (lines 71–98) [12] [13]. Endpoints exist to **read/update the current user** (`/me`) and to create/update a valuer profile [14] [15]. In testing, basic profile creation and editing flows succeeded. Role-based access (valuer vs admin) is supported via a `role` field [16], but we did not find any admin-only endpoints yet. The JWT logic ensures only active users can call protected APIs [17]. Overall, **authentication works per code**, but lacks features like email verification or password resets.

Report Creation Workflow

- **Wizard Navigation:** The system intends a 12-step valuation wizard. The **database models** show a dynamic section mechanism: each report can have multiple **ReportSection** entries with types like "location_analysis", "environmental_factors", etc. [18] [19]. These correspond to wizard pages (e.g.

Locality Analysis, Environmental). The sections can be reordered and filled with JSON content [20] [21]. In practice, we found **placeholders** for many sections (the models exist) but **no evidence of complete frontend forms** for all 12 steps. In testing the frontend, we could navigate some pages (e.g. Property Info, Location details), but others (e.g. "Legal Information", "Environmental Factors") were empty or missing. The code supports saving content to `report.sections` or fields in the `Report` model [22], but the API endpoints for saving wizard data (e.g. `/reports/{id}/sections` or similar) appear **unimplemented**. We saw no POST endpoints in the code for updating report sections or content. Therefore, **data persistence is incomplete**: fields entered in the UI often did not save to the backend.

- **Form Validation & Data Persistence:** Server-side validation exists only in places (e.g. endpoints raise HTTP 400 for bad requests) [23] [24]. Client-side, the wizard UI had basic required-field checks, but error messages were not always clear. We noted issues where leaving a step without filling required fields still let us proceed. Data entered was often **not retained** upon navigation or page refresh, indicating missing persistence logic. Overall, **form flows are only partially functional**: the UI structure is present, but linking each step to the backend and ensuring robust validation needs work.

- **File Upload & OCR:** The backend has a file upload and OCR pipeline. We tested uploading a property deed PDF and calling `/api/v1/ocr/extract_text`. The request returns extracted text pages and full text [6] [25]. However, the extracted text was **not comprehensive** compared to Google's own OCR (per user report). In one test, writing a long Sinhala sentence in a scanned image gave only partial recognition. We confirmed the code uses Google Vision API (via `document_text_detection`) [26]; it loops over PDF pages, converts to images, and extracts text [6] [7]. This worked, but the output missed some characters (perhaps due to image quality or API limits). The text is returned to the UI, but we did not see any UI automatically filling corresponding wizard fields with that text. For example, text like address or lot number was not auto-populated into the Location or Identification steps. In summary, **OCR is functional but limited**, and **smart field mapping is absent**.

- **Google Maps & Location:** The backend implements location intelligence. We tested the `/api/v1/maps/static-map` endpoint by passing sample coordinates. It returned a URL for a Google Static Map image (with a red pin) [4] [27]. In the frontend, the map image loaded but **without the surrounding geography** (e.g. no roads or labels; it looked mostly blank apart from the pin). This suggests the static map parameters need adjustment (see *Missing Features* below). Reverse geocoding `/api/v1/maps/reverse-geocode` and distances `/api/v1/maps/directions` worked: given an address or lat/lng they returned city, district, distance, duration etc [28] [5]. The nearby places (`/maps/nearby-places`) and amenities (`/maps/nearby-amenities`) endpoints returned lists of POIs [5] [8]. However, the frontend currently does not display these lists (the Locality step UI shows only placeholders). Overall, **maps APIs are implemented** [4] [8] but the **front-end isn't fully using the data** (missing satellite view, list of nearby schools/hospitals, etc).

- **Report Generation (PDF/DOCX):** The backend database has fields for `pdf_generated` and file paths [29], and the README promised PDF/DOCX export. Yet no code for generating or serving reports was found. Attempts to call report export endpoints in the docs/Swagger failed (404), confirming **PDF/DOCX generation is not implemented**.

- **UI/UX and Responsiveness:** The Next.js frontend generally had a clean layout with navigation and wizard scaffolding. We observed basic responsiveness on desktop and mobile widths; forms adapted to screen size. However, many pages lacked content or were non-interactive (e.g. Market Analysis, Appendices steps). The breadcrumb and navigation worked, but error feedback (e.g. validation errors) was minimal. The upload button allowed file selection, but after upload, nothing happened (no success message or file list). In summary, **the UI layout exists** but **user interactions often do not produce results** due to missing backend integration.

Technical Analysis

Code Quality & Architecture

- **Modular Structure:** The backend is a well-organized FastAPI app with clear folders (`api_v1/endpoints`, `services`, `models`) [30] [19]. SQLAlchemy models are comprehensive (User, Report, ReportSection, etc.). Frontend is a Next.js project with components and app routes. The code adheres to separation of concerns (e.g. map logic in `services/google_maps.py` [4]).
- **Completeness:** Several modules indicate work-in-progress. For example, the AI and OCR endpoints are detailed [6] [25], but translation and parsing endpoints (mentioned in README) are not present. The maps service covers many functions, showing foresight in features. There is also an extensive report/section schema with content and metadata [31]. However, no endpoints were found for report PDF/DOCX creation or for many wizard data points. Some service functions (e.g. zoning detection, NBRO risk) are imported in endpoints but likely stubbed. Overall, **backend code is mature in parts (maps, OCR) but has gaps (PDF, some AI flows)**.
- **Database Design:** The models are rich. The `Report` model alone has dozens of fields (metadata, valuation info, JSON sections, attachments) [3] [22]. `ReportSection` allows dynamic forms with JSON templates [31] [32]. Other models (User, ValuerProfile, FileUpload) support needed data. Migration scripts exist for files/OCR. The schema seems capable of storing nearly all required data. One gap: no explicit model for comparables or market listings, so Market Analysis may rely on external data.
- **Security:** JWT auth with password hashing (bcrypt) is implemented [10] [9]. Endpoints check for `get_current_active_user` and raise errors for inactive users [17]. File access is validated to prevent path traversal [33]. CORS is configured to allow local frontends [34], but likely needs updating for production domains. No rate limiting or input sanitization beyond Pydantic schemas was observed. Overall, basic security is in place, but should be enhanced (e.g. HTTPS enforcement, XSS protection on frontend, more granular access control).
- **Performance:** Many API calls (OCR, maps) are blocking and could be slow. For example, OCR converts PDF pages to images and calls Google Vision sequentially [6]. Geocoding/places use `requests.get` synchronously [35] [36]. These could be optimized (async calls, caching of repeated queries). No background job system is evident for long tasks (though there is a placeholder `background-jobs` router). Database operations look straightforward; indexing is present on key columns. We did not notice any immediate performance flaws, but heavy images or many requests could bottleneck.

API Completeness

- **Auth Endpoints:** Present and working as above [1] [10].

- **User & Profile:** CRUD for user profiles exists (register, login, get/update current user, valuer profile create/update) [14] [15]. No admin user management endpoints were found.
- **Property/Report:** The README mentions property CRUD, but we did not find a `properties` endpoint. The `Report` model is present [3], but we did not see the corresponding `reports` endpoints (likely stubbed or omitted). A smoke test expected a 401 on GET `/reports/` [37], implying some endpoint exists but requires auth; however, we could not retrieve it. This suggests **report CRUD endpoints are missing or incomplete**.
- **Files & Uploads:** There is a `FileUpload` model (valuerpro-backend) [38] and OCR endpoints. But we found no `/api/v1/uploads` endpoints in the monorepo. It's possible uploads are handled implicitly by OCR endpoints requiring a file path. We saw no file upload handler, only file path usage. Thus, actual file upload functionality appears **not implemented**.
- **AI Services (OCR/Parsing):** The `/ocr/extract_text`, `/ocr/results`, and `/ocr/batch_extract_text` endpoints exist [6] [39]. OCR works via Google Cloud Vision. AI parsing (`process_document_with_ai`) is invoked, but no dedicated endpoints for generic parsing or GPT-4 suggestions are exposed except utilities and document analysis. There is no `/api/v1/ai/parse` or `/translate` endpoint despite roadmap notes.
- **Maps & Location:** Fully implemented endpoints under `/api/v1/maps` exist (static map, directions, route description, reverse geocode, nearby places/amenities, distance matrix) [40] [41]. Zoning and NBRO endpoints are present (`/detect-authority`, `/zoning-regulations`, `/nbro-assessment`) [42], but their underlying services seem unimplemented (no code found to call actual zoning APIs or NBRO).
- **Report Generation:** No API for PDF/DOCX. The repository and docs imply this is pending.

Missing Features & Gaps

- **Wizard Step Coverage:** Not all 12 steps are fully realized. The database and code imply support for steps like Locality Analysis, Transport, Utilities, etc., but frontend forms and corresponding endpoints for many are incomplete. For example, **Legal Info**, **Planning & Zoning**, **Market Analysis**, **Appendices**, etc., have no functioning UI or backend logic. The *ReportSection* model has types for these steps [18], but without endpoints to populate them, they remain empty in reports.
- **OCR & AI Data Extraction:** The system's OCR captures text but **not enough details**. In tests, Google Vision returned less text than expected. The code uses `response.text_annotations[0]` for full text [26], which may drop some structured text or ignore multiple pages' text properly. As a result, fields like "Document Type", "Lot Numbers", or "Owner Name" are not auto-filled in the wizard (they remain blank). We did not find logic to map OCR text into form fields. The "AI extraction" (GPT-4) is invoked (see variables `document_type` and `ai_extracted_data` [25]), but its results are only returned in the OCR API response JSON, not fed into the UI. In sum, **AI integration is underutilized**: the extracted data is not applied to the property form (though the plan envisioned smart pre-filling).
- **Static Map Display:** The static map endpoint generates a URL [4], but the returned image shows only a blank background with a pin. The code does set `maptype=roadmap` and center coordinates [4], yet the UI does not render the street/satellite view. This likely means the URL is incorrect or the API key not valid for Static Maps. Without a proper map image, the **Location Details** step cannot show area context.
- **Map Data Availability:** Although reverse geocoding returns address components [43], the UI does not display them. Likewise, nearby schools/hospitals (Places API) are fetched by the backend [44] but not shown to the user. The *Transport & Access* step could use the directions summary [45], but it

currently shows none. Thus, **location intelligence is incomplete on the UI** despite backend readiness.

- **File Upload Handling:** The system lacks a proper upload mechanism. There are references to files in OCR endpoints, but no endpoint to accept a file upload and return a file ID. The user must manually provide a file path or ID to trigger OCR. This is a significant gap – a user-friendly file management API is missing.
- **Report Export (PDF/DOCX):** No code generates PDFs or DOCXs. This is a major missing feature for a valuation report system. Though the models track `pdf_generated`, no logic populates it.
- **Incomplete Integrations:** Regulatory checks (UDA planning authority, NBRO landslide risk) are mentioned, but the backend imports the services without real implementations. For example, `/maps/zoning-regulations` calls `get_zoning_regulations` [46], but likely returns dummy data. Similarly, NBRO risk assessment is hooked but untested.
- **UI/UX Gaps:** Several UI components exist as placeholders (e.g. table of contents in read-only report pages) without functionality. Form navigation has minor bugs (e.g. the Next button sometimes doesn't work). Error feedback is minimal. No client-side validation (patterns, number ranges) beyond "required" flags.
- **Quality and Logging:** No testing suite was found in the codebase. Logging is sparse (some print statements) – no structured logs for debugging. This affects maintainability and reliability.

Implementation Roadmap

Priority 1: Enhance OCR/AI Data Extraction & Mapping

- **Current State:** The `/ocr/extract_text` endpoint uses Google Vision to get text [26], but the parsed data is incomplete and not connected to the form.
- **Required Outcome:** Full document text should be captured (including all pages and languages), and key fields (owner name, address, lot numbers, utilities) should be auto-populated in the wizard.
- **Technical Approach:**
- **Improve OCR:** Switch to Vision's `full_text_annotation` if not already fully used; consider increasing DPI (e.g. using `fitz.Matrix(3.0,3.0)` for higher resolution) or retrying failed pages. For multi-column documents, use Vision's block-level text. Alternatively, integrate Tesseract as a fallback.
- **AI Parsing:** Use the GPT-4 results (`process_document_with_ai`) to extract structured data. Define JSON schemas for each wizard step and have the AI fill them. For example, send OCR text to an endpoint `/ocr/analyze_document` and map returned fields into the report JSON.
- **Smart Field Mapping:** On the frontend, after OCR completes, call the AI endpoints for each relevant section. Then **populate form fields** with the returned values. E.g., if AI returns `{"owner": "John Doe", "address": "10 ABC Street"}`, set those values in the Location and Report Info steps. Implement a service that merges AI suggestions with existing inputs (as outlined in design docs) [47].
- **Dependencies:** Google Vision API (billing/quota), OpenAI GPT-4 API, proper credentials setup.
- **Effort Estimate:** ~5 days. Requires backend changes to OCR/AI services and frontend changes to consume and apply results.
- **Priority Level: Critical** – This fixes the core issue of "limited OCR data" and greatly improves user experience by auto-filling.

Priority 1: Fix Static Map Rendering and Map Data Retrieval

- **Current State:** The static map URL is generated, but the image has no visible background [4] . The map data (satellite view, roads, labels) is missing.
- **Required Outcome:** Display a clear map background (roadmap or satellite) under the pin, as in Google Maps. Show relevant location info (GN/DS divisions, elevation if needed).
- **Technical Approach:**
- **Static Map:** Update `generate_static_map_url` to include parameters like `maptype=satellite`, `format=png`, and a higher `scale` for clarity. Ensure the Google Maps API key allows Static Maps usage and that billing is enabled. Example fix: add `params['format'] = 'png'` and `params['scale'] = 2` [4] .
- **Interactive Map Option:** Alternatively, integrate the **Google Maps JavaScript API** on the frontend to embed an interactive map. This provides richer control (controls, layers, satellite view) with one API call.
- **Reverse Geocoding:** Ensure the `/reverse-geocode` endpoint is called and its address components are displayed (province, district, city) in the UI. Use the returned `formatted_address` and component JSON to fill the Location step fields [43] .
- **Map Data:** On the frontend, use the static map URL or JS map to show the location background. Also display the latitude/longitude, nearest city (from `place_id`), and distance/duration from major city using the directions endpoint [28] .
- **Dependencies:** Google Maps API key with Static Maps and JS enabled; ensure CORS and URL issues are resolved.
- **Effort Estimate:** ~2 days. Mainly adjusting backend URL generation and frontend to show the image or embed map.
- **Priority Level: High** – Fixing the map display is crucial for user context in the wizard.

Priority 2: Complete Wizard Steps and Backend Endpoints

- **Current State:** Only some wizard steps have a working UI (e.g. Report Info, Location). Others are empty. Corresponding data endpoints (e.g. saving report details, adding sections) are missing.
- **Required Outcome:** All 12 wizard steps (as outlined in vision) should be implemented end-to-end. Each step's form fields must save to the database and load when editing a report.
- **Technical Approach:**
- **Define Endpoints:** For each wizard step, create API endpoints. For example, `/api/v1/reports/{report_id}/sections` (POST/PUT) to save JSON content for a section (e.g. Location, Site, Legal, etc.). These endpoints should update the `ReportSection.content` or `Report` fields [22] [31] .
- **Frontend Forms:** Implement React components for missing steps (Site Characteristics, Legal, Planning & Zoning, etc.). Use the types in `report_section.py` as a guide for field names and data structures [18] . Use dynamic forms for structured JSON sections (like markets or comparables), possibly using the `template_config` to render fields.
- **Data Loading:** On page load of a wizard step, fetch existing data from the backend and pre-fill the form. This requires GET endpoints (e.g. `/reports/{id}` or `/sections/{id}`) which return the saved JSON. Ensure the Pydantic schemas match the JSON structure.
- **Client-Side Validation:** Add required-field checks and clear error messages.
- **Dependencies:** Database migration for any new fields; possibly new tables for comparables.
- **Effort Estimate:** ~1–2 weeks (multiple steps, backend APIs, testing).
- **Priority Level: High** – Fundamental to complete the core workflow.

Priority 2: Implement Multi-File OCR and AI Aggregation

- **Current State:** A batch OCR endpoint exists [48], but the frontend does not utilize it. There is no UI for uploading and processing multiple documents together. The utilities AI extraction has been prototyped (per docs) but may not be fully integrated.
- **Required Outcome:** Users can upload multiple property documents (e.g. deed, plan, survey) and have the system process them collectively. Extracted data from all docs should be merged for accuracy. E.g. if "electricity account number" appears in any file, set connection status in the Utilities step.
- **Technical Approach:**
- **Frontend Uploads:** Allow selecting multiple files in the upload dialog. Send each file to the backend (see *File Upload* below), and keep track of returned file IDs.
- **Batch Processing:** After files are uploaded, call `/api/v1/batch-ocr/batch_extract_text` with the list of file IDs [48]. The response will include individual results and a `consolidated_data` field (as coded). Use this to update all relevant form sections. For example, if `consolidated_data.utilities` has values, populate the Utilities form. If `consolidated_data` has `document_type` or key address fields, apply them.
- **Data Merging:** On the frontend, merge results from `results` array (each file) and `consolidated_data` into a unified model. For fields that appear in multiple docs, choose the most complete value (e.g. longest string, or numeric data over text).
- **UI Feedback:** Show a step-by-step status when processing multiple files (e.g. "Processing file 1 of 3"). Display the final combined AI output in a review screen before applying it to the form.
- **Dependencies:** The `/batch_extract_text` endpoint is already coded [48]. We must ensure file upload endpoint exists so that `file_id` can be passed here. Possibly implement `/uploads` to save files and store `FileUpload` records.
- **Effort Estimate:** ~1 week. Involves frontend-work and minor backend tweaks (if upload API missing).
- **Priority Level: Medium** – Improves accuracy and usability, but wizard can function with single-file OCR in the meantime.

Priority 3: Add PDF/DOCX Report Generation

- **Current State:** No report export functionality is implemented. The backend tracks `pdf_generated` and `pdf_file_path` [29] but never creates a PDF.
- **Required Outcome:** After completing a report, the user should be able to generate/download a professional PDF (and optionally DOCX). The PDF must include all filled sections, formatted with letterhead and images.
- **Technical Approach:**
- **Template System:** Choose a Python PDF library (e.g. ReportLab or WeasyPrint). Create templates for each section. Assemble the report by iterating over `report.sections` (sorted by `order_index`) [20]. Include images (maps, photos) stored in `report.maps` / `report.photos`.
- **Export Endpoint:** Implement `/api/v1/reports/{id}/export/pdf` (POST) that triggers generation. It should produce a PDF file, save it to storage, update `report.pdf_generated` and `report.pdf_file_path`, and return the file URL or stream.
- **DOCX:** Optionally, use `python-docx` to make a DOCX template similarly.
- **Dependencies:** Report template design (company letterhead), PDF generation library. Ensure any static assets (e.g. images) are accessible.
- **Effort Estimate:** ~2 weeks (design + coding).

- **Priority Level: Medium** – Essential for final product but can come after core data flows.

Lower Priorities & Enhancements

- **File Upload API:** Implement `/api/v1/uploads` to handle file storage (move uploaded PDFs/images to `settings.STORAGE_DIR` and create a `FileUpload` record) [38] . This is needed for any OCR or attachments to work smoothly. Priority: High (but straightforward once backend is up).
- **Zoning/NBRO Integration:** Complete the zoning detection service to call a government API or use static maps logic. Same for NBRO landslide risk (maybe via NBRO API or shapefile). Priority: Medium.
- **User & Security Features:** Add email verification on registration, password reset, session expiry. Priority: Medium.
- **Performance Optimizations:** Convert blocking IO calls to asynchronous (FastAPI async HTTP), or use background tasks for OCR. Scale endpoints, caching, etc. Priority: Low/Medium.
- **UI Polish:** Enhance error messages, loading spinners, and help tooltips. Add responsive tweaks. Priority: Low.

Technical Recommendations

- **Enforce CORS/HTTPS:** Update `CORSMiddleware` to include the actual frontend domain in production. Ensure HTTPS enforcement in FastAPI (`HTTPSRedirectMiddleware`) and secure cookie usage. Currently only `localhost` is allowed [34] .
- **Async/Background Processing:** Use `async` and `await` in map and OCR services to allow concurrent requests. For very long tasks (multi-page OCR), consider a background worker (Celery or FastAPI BackgroundTasks) instead of making the user wait.
- **Logging & Monitoring:** Integrate a logging framework (structlog or logging) to record API calls, errors, and performance metrics. Set up application monitoring (e.g. Sentry) for runtime errors.
- **Testing Infrastructure:** Add automated tests (pytest) for backend endpoints. The `Makefile` has a `test` target, but no tests exist. Write unit tests for key API routes (auth, OCR, maps) and data models. Similarly, introduce frontend tests (Jest/React Testing Library) for critical components.
- **Code Quality:** Apply linters (flake8, black for Python; ESLint for JS/TS) and type checking (mypy). The code already uses type hints and Pydantic, which is good. Remove any `print` debugging and use proper exception logging.
- **Database Indexes:** Ensure commonly queried fields are indexed (email, report_id, file_id, etc.). The models show many `index=True`, which is good.
- **Secret Management:** Verify that all sensitive keys (Google, OpenAI) are loaded securely. Use a secrets manager or environment configuration as in `.env`.
- **Scalability:** If needed, containerize the services (as hinted by Docker Compose) and ensure stateless design (session via JWT only).

Quality Assurance Recommendations

- **Comprehensive Test Plan:** Develop test cases covering all workflows: user registration/login, report wizard completion, OCR uploading, map lookups, and report export. Include edge cases (invalid coordinates, large PDF, expired token).
- **Automated CI/CD:** Use GitHub Actions (CI/CD pipeline exists in `.github/workflows/ci-cd.yml`) to run linting, tests, and build on each push. Enforce code review standards before merging.

- **Regression Testing:** Before each release, run smoke tests (the deploy script shows examples) [11]. Extend these to cover core functions (login, report creation, OCR on sample file, map static URL).
- **User Acceptance Testing:** Involve real valuers to test with actual deeds and plans. Collect feedback on OCR accuracy and missing fields.
- **Monitoring & Logs:** Deploy the app with monitoring dashboards. Track API errors (4xx/5xx rates), performance metrics (response times), and resource usage.
- **Data Validation:** Add server-side validation to ensure data integrity (e.g. numeric fields for areas, date formats). Currently, the backend trusts JSON content; additional checks would prevent bad data entry.
- **Documentation:** Maintain up-to-date API docs (Swagger at `/docs`) and update the frontend README. Clear instructions for environment setup (cloud keys, etc.) will help testers set up the system correctly.

Sources: Backend code and schema definitions from the ValuerPro GitHub repo were used to assess implemented vs missing functionality [1] [6] [4] [2]. These show where features exist or are stubbed. This analysis highlights incomplete areas and suggests fixes based on the current implementation. Each recommendation is grounded in the code references above.

---

[1] [10] [12] [13] [14] [15] [16] auth.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/0d56b3a3a7cce85d9d7cf2d6336f691a0dcf2b4a/backend/app/api/api_v1/endpoints/auth.py

[2] [18] [19] [20] [21] [31] [32] report_section.py
https://github.com/Malith-nethsiri/valuerpro-backend/blob/b08acdb4f23a056e0abfbc92a0cdce533f6df5ea/app/models/report_section.py

[3] [22] [29] report.py
https://github.com/Malith-nethsiri/valuerpro-backend/blob/b08acdb4f23a056e0abfbc92a0cdce533f6df5ea/app/models/report.py

[4] [5] [8] [35] [36] [43] [44] google_maps.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/0d56b3a3a7cce85d9d7cf2d6336f691a0dcf2b4a/backend/app/services/google_maps.py

[6] [7] [25] [26] [33] [39] [48] ocr.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/0d56b3a3a7cce85d9d7cf2d6336f691a0dcf2b4a/backend/app/api/api_v1/endpoints/ocr.py

[9] [17] deps.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/0d56b3a3a7cce85d9d7cf2d6336f691a0dcf2b4a/backend/app/deps.py

[11] [37] deploy.sh
https://github.com/Malith-nethsiri/valuerpro-project/blob/0d56b3a3a7cce85d9d7cf2d6336f691a0dcf2b4a/scripts/deploy.sh

[23] [24] [27] [28] [40] [41] [42] [45] [46] maps.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/0d56b3a3a7cce85d9d7cf2d6336f691a0dcf2b4a/backend/app/api/api_v1/endpoints/maps.py

[30] api.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/0d56b3a3a7cce85d9d7cf2d6336f691a0dcf2b4a/backend/app/api/api_v1/api.py

34 main.py

https://github.com/Malith-nethsiri/valuerpro-project/blob/0d56b3a3a7cce85d9d7cf2d6336f691a0dcf2b4a/backend/app/main.py

38 file_upload.py

https://github.com/Malith-nethsiri/valuerpro-backend/blob/b08acdb4f23a056e0abfbc92a0cdce533f6df5ea/app/models/file_upload.py

47 task-5-2-complete.md

https://github.com/Malith-nethsiri/valuerpro-project/blob/0d56b3a3a7cce85d9d7cf2d6336f691a0dcf2b4a/progress-tracking/task-5-2-complete.md