ChatGPT

ValuerPro Code Review – Comprehensive File-by-File Analysis

Project Overview: ValuerPro is an AI-driven property valuation system with a Next.js frontend and FastAPI backend [1]. The codebase is ambitious and feature-rich – including a 15-step valuation wizard, OCR/AI document analysis, Google Maps integration, and robust PDF/DOCX report generation. Below, we review *every major file* in the project, detailing its **good aspects**, **issues/concerns**, and **suggested improvements** from an expert perspective.

Documentation & Configuration Files

README.md – Project Introduction & Setup

Good: The README is very detailed and well-structured. It clearly outlines the project features, architecture, and quick-start steps [2] [3]. The inclusion of an architecture diagram (file structure) and key workflow breakdown is excellent for newcomers [4] [5]. It also mentions security/compliance features and how to deploy to production, which shows foresight.

Bad: The README could possibly become out-of-sync with the code as the project evolves (e.g. it mentions "OpenAI GPT-4o" which might confuse readers – perhaps a typo of GPT-4) [6]. A minor issue is that the quick-start assumes a certain environment (e.g. Google Cloud credentials) without linking to more detailed config instructions.

Improvements: Keep the README updated alongside code changes – for instance, clarify the OpenAI model naming and ensure any future config changes (new env vars, services) are documented. Including a brief summary of how to run tests or pointing to a `CONTRIBUTING.md` would further strengthen the documentation.

Makefile – Development Conveniences

Good: The Makefile provides handy shortcuts for common tasks (running backend, frontend, DB, tests, etc.) [7] [8]. This improves developer experience by documenting how to start each service and even includes a combined `dev:all` target that waits for the DB to be ready [9]. The presence of targets for migrations and cleanup is thoughtful.

Bad: Some commands (like `dev:backend` using `uvicorn --reload`) are fine for development but might not reflect how production runs (which is okay as long as it's understood). There's a small assumption that `docker compose` is installed (the syntax uses both `docker-compose` and `docker compose` interchangeably, which could be inconsistent across environments – but here it seems consistent).

Improvements: The Makefile could add an explicit `prod` or `build` target to build Docker images or run the production compose, complementing the `scripts/deploy.sh`. Also, adding comments for any non-obvious steps (like the `timeout` logic for DB readiness [10]) can help new developers understand the rationale.

CLAUDE.md – AI Assistant Planning

Good: This file appears to capture initial project instructions for an AI assistant (possibly used during development). It lists non-negotiable requirements (monorepo structure, tech stack, no payments, etc.) and dev rules [11] [12]. This shows the team carefully planned the architecture from the start, ensuring key decisions (like JWT auth, secure cookies, environment variables) were documented.

Bad: It's not directly relevant to the application's runtime or user docs. As such, it might confuse contributors who aren't aware of its context (it references `./plan.md` which isn't in the repo and rules like "Ask before deleting files" likely applied to an AI tool, not humans).

Improvements: Consider moving such AI prompt files to a separate `docs/` or clearly mark them as historical planning notes. This avoids confusion for future maintainers. No changes to code needed – just documentation organization.

SECURITY.md – Security Guidelines

Good: The SECURITY.md details the project's security posture and likely provides guidelines on reporting vulnerabilities. (Though not quoted here, we assume it covers best practices given the security-conscious code in the project – e.g. use of security headers and rate limiting in code). Its presence indicates the team values security and likely outlines password policies, data protection, etc.

Bad: If SECURITY.md is generic or not kept up-to-date with code (for example, if it doesn't mention the rate limiting or JWT approach actually used), it can become stale. Security documentation must reflect the implementation.

Improvements: Ensure SECURITY.md aligns with the code – e.g., mention the use of `TrustedHostMiddleware` and secure cookies if applicable. It could also include a checklist of security features (like whether encryption is used for sensitive data). Keeping this file updated will help external auditors or contributors quickly gauge security aspects.

PRIVACY_POLICY.md and TERMS_OF_SERVICE.md

Good: Including a Privacy Policy and Terms shows professionalism and compliance (especially since the app processes potentially sensitive property data). These files likely cover how user data (like uploaded documents) is handled, fulfilling legal requirements.

Bad: From a development perspective, these files don't affect code, but if the application's data handling changes (e.g., adding third-party services), the policies might need updates. No technical bad points beyond the need for maintenance.

Improvements: Align these documents with actual functionality – for instance, if using Google Cloud Vision, ensure the privacy policy discloses that user files are sent to Google's API. Periodically review them for compliance as features expand.

.github/workflows/ci-cd.yml – CI/CD Pipeline

Good: The CI configuration is extremely thorough. It runs linting (Black, isort, Flake8) [13], type checks (mypy), security scans (Bandit, Safety) [14], and a full battery of tests (unit tests, property-based tests, performance benchmarks, even AI-generated edge-case tests) [15]. Similarly, it sets up Node for frontend tests (unit, e2e with Playwright, visual regression, accessibility audits) [16] [17]. This automated quality assurance is a major positive – it catches errors early and enforces code standards. Artifacts like coverage and reports are uploaded for inspection [18] [19].

Bad: The pipeline is complex – which can slow down CI runs. For instance, launching both backend and frontend to run Playwright tests is great for coverage but might be time-consuming. Also, the deploy steps in the workflow (if any) are placeholders (just echoing deploy messages) [20] [21]. This suggests deployment is not fully automated yet.

Improvements: The CI config is solid. One improvement could be splitting jobs (so lint/type-check can run in parallel with tests to speed things up). Also, once deployment steps are ready, replace the placeholder `echo "Deploying..."` with actual scripts or remove them until used to avoid confusion. Overall, maintain this CI as the project grows – it's an asset for code quality.

Backend – Core Application Code

app/core/config.py – Configuration Management

Good: The config uses pydantic settings (though not explicitly shown in the snippet, the code references `settings` throughout, likely defined in config.py). This centralized approach is good for managing environment variables (e.g., `GOOGLE_CLOUD_CREDENTIALS_PATH`, `SECRET_KEY`, etc.) and feature flags like DEBUG or RATE_LIMIT_ENABLED [22] [23]. Having things like `APP_VERSION`, `CORS_ORIGINS`, etc., in config with sensible defaults is a strong practice.

Bad: If any secrets (like API keys) are defaulted in config.py, that would be risky – but presumably, they are pulled from environment. One minor concern: ensuring that all config values (like log file paths, storage directories) exist and are writable. The code does attempt to create directories on startup [24], which is good.

Improvements: Use pydantic's `.env` file support to document all required env vars in a `.env.example`. This would make setup easier. Additionally, consider grouping related settings into BaseSettings subclasses (for example, a DatabaseSettings, SecuritySettings, etc.) for clarity – though for this project size, a single `settings` object is fine.

app/main.py – FastAPI App Initialization

Good: This file is a highlight of best practices. It configures logging and lifespan events, and includes a host of middleware for security and performance: - **Security headers** via `SecurityMiddleware` (likely sets HSTS, etc.) and rate limiting via `RateLimitMiddleware` [25] [26]. - **CORS** is configured thoughtfully: in development it allows all origins, but in production it restricts to specific allowed hosts from settings [27] [28]. - **TrustedHostMiddleware** is used to prevent Host header attacks [29]. - GZip compression is enabled for responses [30]. - A global request timing middleware adds an `X-Process-Time` header and logs slow requests (over 1s) [31]. This is great for monitoring performance.
- Exception handlers are defined for validation errors and generic exceptions, returning structured JSON with details and timestamps [32] [33]. This improves API clarity by providing helpful error messages (e.g., listing validation issues) [34]. The generic handler even differentiates between dev and prod, exposing error detail only in debug mode [35] [36] – an excellent security practice to avoid leaking internals in production.

Bad: Very few issues here. One observation: the `add_process_time_header` middleware uses `time.time()` which is fine, though using `asyncio.get_event_loop().time()` might be slightly more precise for event loop time – not a big deal. Also, ensure that the SecurityMiddleware and RateLimitMiddleware are implemented correctly in `app/middleware/*` (we assume they add headers and throttle clients; if they aren't optimized, they could be a bottleneck).

Improvements: Monitor the performance impact of the middleware stack in production – e.g., rate limiting each request can add overhead. If needed, tune the limits or allow bypassing certain endpoints (like health checks) from heavy middleware. Additionally, consider logging to a file or external system instead of basicConfig (though basicConfig is acceptable for simplicity). Overall, main.py demonstrates clean design; just keep it updated as new middleware or routers are added.

app/db.py – Database Session Management

Good: (Though not shown above, likely this file sets up SQLAlchemy session Local and engine). It likely provides `SessionLocal` and a dependency `get_db` for FastAPI to inject DB sessions [37] [38]. This pattern (opening a session per request and closing it) is standard and helps avoid connection leaks.

Bad: If `db.py` is minimal, there's not much to criticize. One thing: if the project uses Alembic, ensure that `db.py`'s engine is used consistently for migrations (the Alembic env file likely imports it). No obvious bad practice if it follows SQLAlchemy docs.

Improvements: If not already, use `sessionmaker` with autoflush=False and autocommit=False for safety. Also, ensure thread-local sessions (if using `scoped_session`) are handled properly in async context (FastAPI's sync dependency might suffice). In short, just maintain the session lifecycle carefully – which the code does via dependency injection.

app/models.py – ORM Model Definitions

Good: The models define a comprehensive schema for the system. Notably: - **User & ValuerProfile:** Splitting professional info into `ValuerProfile` is smart. The one-to-one relationship (valuer_profile uselist=False) on User makes it easy to extend user data without cluttering the main table [39] [40]. The use of UUID primary keys everywhere is great for uniqueness. Relationships are set up to cascade logically (User

-> Reports, Reports -> Properties, etc.), enabling ORM traversals [41] [42]. - **Report & related:** The Report model tracks status, dates, and foreign keys to author and client [43] [44]. The unique index on (author_id, ref) ensures each user's report reference number is unique [45] – a thoughtful business rule to prevent duplicate report IDs [46] [47]. - **Property and Steps:** The breakdown of property data into multiple tables (Identification, Location, Access, Site, Building(s), Utilities, Planning, Locality, ValuationLines, ValuationSummary, etc.) is very granular [48] [49] [50] [51]. This mirrors the wizard steps. The benefit is strong normalization: each aspect of the valuation is stored in a structured way (e.g., boundaries JSON in Identification, building materials in Building, etc.). This is good for ensuring data integrity and flexible querying (e.g., one could query all properties in a certain district via Location.district).
- **Other models:** OCRResult, AISuggestion, Revision, Template, RegulationDocument, etc., cover advanced features [52] [53] [54] [55]. For example, OCRResult stores extracted text and confidence [56], and AISuggestion can hold AI-proposed content for report sections [57]. The design anticipates future features like versioning (Revision diff) and regulation libraries – impressive forethought. - Using `ARRAY` and `JSON` columns (PostgreSQL-specific) for lists and structured data is a good choice where appropriate (e.g., qualifications list, boundaries JSON) [58] [59].

Bad: The flip side of such normalization is complexity. The 15-step wizard data is spread across many tables, which means multiple joins to reconstruct a full report. For example, generating the final report might need to join Property -> Identification, Location, Access, Site, Buildings, etc. If not handled carefully, this could impact performance. Also, creating a new report with all sub-objects requires multiple inserts – the code seems to handle some of this in endpoints (e.g., `createProperty` API and separate endpoints for adding lines, etc.). There's also some duplication risk: e.g., both `ValuationSummary` and `ValuationLines` store values, so keeping them in sync is important.

Improvements: Consider convenience methods or ORM relationships to quickly fetch all related data. For instance, a `Report.complete()` method to gather its properties and all sub-entries in one go. If performance becomes an issue, one might consider caching or denormalizing some data (like storing a computed `completion_percentage` or summary in the Report). Additionally, ensure database indices are added for any frequently queried fields (the code does add some, like index on Report.ref and maybe implicitly via ForeignKeys). Lastly, writing thorough migration scripts (with Alembic, which the project includes) for any model changes will be crucial given the model count.

app/schemas.py – Pydantic Models (API Schemas)

Good: Pydantic schemas mirror the ORM models for input/output validation. The use of Pydantic's `ConfigDict(from_attributes=True)` allows easy conversion from ORM objects to schema [60] [61]. The schemas define validators for various fields, leveraging the custom validation functions in `app.core.validation` – for example, `validate_future_date` ensures dates are not too far ahead [62], and choices are validated against allowed values [63] [64]. This is excellent – it catches invalid data at the API boundary, maintaining data integrity (e.g., ensuring `report.status` is one of allowed choices, currency is valid, etc.). The presence of `ValidationMixin.clean_strings` means schemas trim whitespace automatically [65], preventing accidental data quality issues like " New York " with trailing spaces.

Bad: Some validators seem a bit arbitrary – e.g., `validate_future_date` only allows up to 30 days in future [66]. If a valuation report is scheduled for months ahead, this could reject a valid date. Such constraints are domain-specific; hopefully they match requirements. Also, `UserCreate` only lightly validates password length ($\geq$8 chars) [67] – it doesn't enforce complexity beyond that. Perhaps the intention is to add more checks (e.g., requiring numbers/symbols) in the `validate_password` (which could use a regex for stronger policies).

Improvements: Fine-tune validation rules based on real-world feedback. For instance, allow more flexibility in date ranges if needed, or enforce stricter password rules for security. Also consider using Pydantic's `model_config` to set `orm_mode=True` (though the use of `from_attributes` achieves similar). Keeping schemas in sync with models as they evolve is important – the structure already looks maintainable, but any model changes need corresponding schema updates.

app/core/validation.py – Custom Validation Utilities

Good: This module is full of useful validators covering positive numbers, percentages, coordinates, phone numbers, postal codes, lot/plan numbers, land extent, currency amounts, date ranges, building areas, etc. [68] [69] [66] [70]. It shows deep domain knowledge (e.g., a Sri Lankan postal code must be 5 digits [71], or an unusually large land extent > 10,000 perches is likely invalid [72]). Encapsulating these rules in functions makes them reusable across schemas and ensures consistency. The presence of these checks significantly increases data quality – e.g., `validate_phone_number` strips common formatting and then enforces 10-15 digits [73], which is great for standardizing contact info.

Bad: One potential drawback is that some limits might be too strict or might change. For example, the 99 million USD max in `validate_currency_amount` [74] or the 100,000 sq ft building area cap [70] – these are reasonable guesses to catch typos, but if a real scenario exceeds them, the system would reject valid data. Essentially, these rules need to be reviewed with real use cases to avoid false positives. Also, some functions raise `ValueError` with messages – Pydantic will capture these and include in response, which is fine, but we must ensure the error messages are user-friendly (they mostly are).

Improvements: Keep these validators configurable where possible. For instance, `validate_future_date` could allow different windows depending on context. Logging when validation fails (for monitoring) might be useful – e.g., if many users trip the 100,000 sq ft rule, maybe the threshold is too low. Also, adding unit tests for each validation function (there may already be tests given the CI) will ensure they work as expected. Overall, this file is very thorough – just maintain it as regulations or standards evolve.

app/core/background_jobs.py – Background Task Queue

Good: This module implements an in-memory job queue with a thread pool to handle long-running tasks (OCR, analysis, PDF generation, etc.) outside the request/response cycle [75] [76]. Good points include: - Clear **Job** data structures (`BackgroundJob` dataclass with fields for status, progress, result, etc. [77] [78] and `JobResult` dataclass for outcome) – this makes it easy to serialize job info to clients via API if needed [79] [80]. - Pre-registered **job handlers** for each job type in `_register_default_handlers` [81]. Each handler is well-defined: e.g., `_handle_ocr_job` calls the OCR logic and updates progress in steps [82], `_handle_analysis_job` calls AI processing [83] [84], `_handle_pdf_generation` and `_handle_docx_generation` call the document service to generate files [85] [86]. These handlers handle fetching necessary data (like loading user/report from DB within the thread) and wrap the operation in try/except to catch failures, marking job as failed if exceptions occur. Progress updates via `_update_progress` (not shown, but clearly used) are great for giving users feedback on long tasks [87] [88]. - **Thread-safety:** There's a lock around job dict access and conditionally cancelling jobs [89] [90] to avoid

race conditions. Using a `ThreadPoolExecutor` with max_workers (4 by default) prevents too many simultaneous jobs hogging resources [91]. - The job logic smartly **reuses existing functions** from the endpoints logic: e.g., `_handle_ocr_job` actually calls the same `extract_text` function from the OCR endpoint, passing in a constructed `OCRRequest` and user info [92]. This avoids duplicating OCR code and ensures consistent behavior whether run immediately or via background.
- There's even a `cleanup_old_jobs` method stub to remove old jobs after 7 days [93] – showing attention to resource cleanup.

Bad: The in-memory approach means if the server restarts, queued jobs are lost. For now, that might be acceptable, but in production a more persistent queue (Redis, Celery, etc.) might be needed for reliability. Also, because the handlers perform database operations and heavy AI calls in threads, one must ensure thread-safety of those libraries (SQLAlchemy sessions are not inherently thread-safe, but here they create a new Session inside the thread for use, which is correct [94]). Another caution: thread pool with 4 workers means at most 4 heavy tasks at once – if each calls external APIs (Vision, OpenAI) or generates PDFs, they might saturate CPU or network. Monitoring will be needed to adjust `max_workers`.

Improvements: If job volume grows, consider abstracting this to use a proper task queue or at least writing job results to the database so they persist across restarts. Also, implement the stubbed features: for example, the `background_jobs.cleanup_old_jobs()` could be called on a schedule to prune the `jobs` dict. Logging each job's completion or failure (the code logs to `logger.info` on success and stores error messages in result) could be extended to notify users via WebSocket or email if needed. For now, the system is fine for moderate workloads – just plan for scaling the job execution in the future.

app/api/api_v1/api.py – Router Inclusion

Good: This file simply creates an APIRouter and includes all the endpoint modules with proper prefixes and tags [95]. It keeps the API modular – each submodule (auth, reports, uploads, ocr, etc.) registers its routes, and this aggregator attaches them under e.g. `/auth`, `/reports`, etc. The presence of a health check here (`/healthz` and `/health/db`) is useful for Kubernetes or uptime monitoring [96]. The database health check actually calls `test_database_connection()` which likely attempts a simple DB query [97] – good for confirming DB connectivity at runtime.

Bad: Nothing significant – this is a clean pattern. Perhaps ensure that tags are consistently named (tags help organize Swagger docs). All included routers have tags defined here, which is fine.

Improvements: If new endpoint modules are added (e.g., if adding a `/admin` API), just remember to include them here. Otherwise, this file is low-maintenance and well-done.

Backend – API Endpoint Modules

auth.py – Authentication & User Management

Good: The auth endpoints cover registration, login, user profile, and valuer profile updates. Highlights: - **Registration**: It checks for existing email to prevent duplicates [98], hashes the password with Bcrypt (via passlib) [99], creates the User and possibly a ValuerProfile if professional info is provided [100] [101]. It wraps profile creation in a try-except so a failure there doesn't abort the whole registration – it logs an error but still returns the user [102]. This fail-safe approach is good (user can still log in even if profile save fails). - **Login**: Uses OAuth2PasswordRequestForm, verifies password using the passlib context [103]. On success, it issues a JWT access token with an expiration [104] [105]. It properly returns `{"access_token": ..., "token_type": "bearer"}` which is standard [106]. - **JWT details**: The token includes an expiry and the subject (user email) and is signed with SECRET_KEY and algorithm from settings [107]. Good use of `jose.jwt` for JWT encoding. - **User self-service**: Endpoints to get current user info (`/me`) and update user (`PUT /me`) are provided [108] [109]. They depend on `get_current_active_user` which likely verifies JWT and user active status. Update logic smartly uses Pydantic's `exclude_unset` to only apply provided fields [110]. - **Profile creation/update**: They allow a logged-in user to create or update their ValuerProfile [111] [112]. The create ensures no profile already exists (so one profile per user) [113]. Update fetches the profile and then sets new values with a loop [114] – simple and effective. - **Security**: Throughout, the code checks that the authenticated user is accessing their own data (e.g., user in `/me` is always the token's user; profile uses current_user.id so it's always self). Also, inactive user handling (if any) would be in `get_current_active_user`.

Bad: Using email as the JWT "sub" claim is okay, but if emails can change, using user ID might be safer. The code uses email likely because that's unique and needed for login – just something to be mindful of (if user changes email, their old token's "sub" is outdated). Also, the login returns only a short-lived access token. There's mention in frontend of a refresh_token (frontend code checks and stores refresh_token if present) [115], but the backend login response doesn't include a refresh token currently (just access_token). So either the frontend is expecting a refresh token that's not implemented, or it's a leftover plan. This discrepancy is minor but worth noting – could confuse the client logic.

Improvements: Consider adding a refresh token mechanism or remove it from frontend if not used. Also, it might be beneficial to log registration events (for audit) or send a welcome email – but that's outside core scope. One security improvement: If brute-force login attempts are a concern, consider rate limiting at the login endpoint (the global RateLimitMiddleware may cover it). Also ensure `ACCESS_TOKEN_EXPIRE_MINUTES` is set reasonably (it's read from settings in code) – a typical value is 15 or 30 minutes for access token life. All in all, the auth.py is solid – just keep token handling in sync with frontend expectations.

reports.py – Report & Client Endpoints

Good: This is a large module covering client management, report CRUD, and various sub-resources (properties, valuation lines, finalization, validation checks, etc.). Good points: - **Client endpoints:** You can create a client (tied to current user as author) [116], list your clients (filtered by your user ID) [117], retrieve and update a client if it belongs to you [118] [119]. This multi-tenancy approach is correct – valuers only see their own clients. The update uses the same schema as create for simplicity (could use a separate update schema to make fields optional, but using Create with all fields required ensures updates always provide complete data) [119]. - **Report creation:** Accepts ReportCreate pydantic model. It prevents duplicate reference numbers by checking the DB first [46] and also catching the unique index IntegrityError on commit [47] – a double safety net. It allows attaching to an existing client_id if provided, ensuring that client belongs to the current user [120] (good authorization check). Defaults are applied for report_date (if not provided, uses now) and status (defaults to "draft") [121]. After creation, it refreshes and returns the report. - **Report listing:** Lists current user's reports, with optional status filter [122] [123]. This is efficient by filtering in the query. - **Report get/update:** The get (`/{id}`) ensures the report exists and belongs to current user [124]. Update (`PUT {id}`) similarly checks and then applies updates from ReportUpdate schema [125] [126] (the code beyond that likely merges fields and saves). These checks enforce that valuers cannot access or modify each other's reports – essential for data isolation. - **Finalize & Validation:** The presence of `/reports/{id}/finalize` is noted and presumably implemented to mark status finalized and timestamp [127] [128]. There are also `/validate` and `pre-finalize-check` endpoints indicated [129] [130] that probably use the `validation_engine` service to assess if a report is complete and identify issues. This is a great feature – it prevents finalizing incomplete reports and gives recommendations (like "fix validation errors" or "address warnings") [131] [132]. - **Properties & Valuation lines**: Although not fully shown above, the code references `create_property` and similar in the frontend API, and the router likely includes sub-routes for adding property details. Given the models, we expect endpoints to create a Property entry for a report, and possibly to add Identification, etc., or to handle them through one combined payload. The design likely leans on the WizardProvider (frontend) saving step-by-step via `reportsAPI.createProperty`, `reportsAPI.createValuationSummary`, etc. [133] [134]. Indeed, in `WizardProvider.saveStepData`, we see cases for identification and valuation that call appropriate `reportsAPI` functions [135] [133]. This modular approach keeps the backend endpoints focused (one endpoint per resource type). - **Appendix generation & document output:** The router imports `document_service`, `validation_engine`, `appendix_generator` at top [136], so presumably there are routes to trigger generating the PDF (`/generate-pdf` perhaps called inside finalize or separate) and to produce appendices for the report. If implemented, this would round out the workflow by allowing the user to get the final documents.

Bad: The main caution is complexity – the reports endpoints have to orchestrate many related pieces. For example, if a report is created without any properties, what is the expected flow? Likely the frontend will immediately call createProperty, etc. The backend should handle partial data gracefully. We notice that `create_report` doesn't automatically create a Property record; that's left to separate calls. This is okay (more flexible), but it means a "report" without any property data is initially empty. The `validation_engine` likely accounts for that and marks completeness accordingly. Also, error handling: if any step fails (like creating a property or valuation summary), does the backend return meaningful errors? The code as given in `WizardProvider.saveReport` and `saveStepData` wraps some calls in try/except and logs errors but doesn't propagate them (e.g., it catches exceptions in `saveStepData` and just logs, not failing the whole wizard save) [137]. This leniency is user-friendly (it won't break the UI if an auto-save fails), but it could hide issues. Logging on the backend for these endpoints will be crucial to debug.

Improvements: Documentation of the API contract for each step would help developers. For instance, clearly stating that to fully create a report, the client must call POST /reports, then POST /reports/{id}/properties, then sub-endpoints for identification, etc. Alternatively, consider a batch endpoint that can accept a full report payload (all steps) for easier integration/testing – though the stepwise approach is fine for the interactive wizard. Another improvement could be to enforce referential integrity through the API: e.g., maybe prevent finalizing a report if no Property exists or if mandatory fields are missing. The presence of the validation endpoints likely covers this, but ensuring the finalize endpoint double-checks `validation_report.can_finalize` before actually finalizing would be wise (so you don't finalize an incomplete report by calling the endpoint directly bypassing UI checks). Lastly, continue writing tests for these endpoints (the repository includes `test_reports.py` which likely covers these flows). Given the complexity, comprehensive tests will ensure everything works together.

ocr.py – OCR Extraction Endpoints

Good: This endpoint handles extracting text from uploaded documents (like survey plans, deeds, images) using Google Cloud Vision. Good aspects: - **Dependency checks:** It attempts to import

`google.cloud.vision`, PyMuPDF (`fitz`), PIL, etc., and sets flags `VISION_AVAILABLE`, `DOCX_AVAILABLE` accordingly [138]. This means the endpoint will gracefully error with a clear message if dependencies aren't installed (instead of crashing) [139]. For instance, if `google.cloud.vision` isn't installed, any call to Vision will return a 400 with "dependencies not installed" – good developer feedback. - **Credential handling:** The helper `get_vision_client()` checks for a credentials path in settings and ensures the environment variable is set, converting relative paths to absolute relative to the backend root [140] [141]. It returns a client or raises a clear HTTP 400 error if credentials are missing or invalid [142] [143]. This prevents attempts to call the Vision API without proper setup. - **PDF handling:** The code can convert PDFs to images page-by-page with PyMuPDF and then OCR each page [144] [145]. It uses a 2x zoom (~150 DPI) for PDF images [146] and the specialized `document_text_detection` for better layout retention [147]. This is great for extracting text from multi-page PDFs while preserving accuracy. It also collects text from each page into `OCRPageResult` objects and ultimately combines them [148] [149]. - **Image handling:** For images, it reads bytes and calls Vision's `text_detection` [150]. The code even handles TIFF by converting to PNG first because Vision might not support multi-page TIFF natively [151] [152]. This attention to different image formats is excellent. - **Preventing injection & access issues:** The function `validate_file_access` ensures the requested file path is within the allowed storage directory and that it exists, and that the extension is one of allowed types [153] [154]. It rejects if someone tried a path traversal or unsupported file type with clear 403/400 errors [155] [154]. This is crucial for security – it ensures users can't OCR arbitrary server files. - **Reuse and caching:** Before doing OCR anew, it checks if an OCR result exists in the DB for that file and returns it immediately to avoid duplicate processing [156]. This improves performance by caching results. - **Translation and AI integration:** After extracting raw text, it checks if translation is available and processes mixed Sinhala/English text via `process_mixed_language_text` [157]. It sets `translation_applied` flag and continues gracefully even if translation fails (it catches exceptions and just proceeds without translation) [158]. Then it passes the text to `process_document_with_ai` to extract structured data (document type, fields, general data) [159]. This hooks the OCR with the AI parsing – so the OCR endpoint not only gets text, but also classifies the document and pulls out key info (lot number, etc.). Storing the `ai_extracted_data` and `document_type` in the OCRResponse is a huge feature – it means front-end gets immediate AI suggestions once OCR is done. - **Saving results:** The code attempts to save the OCRResult to DB (raw text, blocks JSON, etc.) if a File record exists [160] [161]. Importantly, it wraps this in a try/except so a DB save failure won't break the whole OCR request [162]. It logs or passes on failure (the code uses `pass`, which ideally would log) to avoid user-facing errors. After processing, it returns an OCRResponse with pages, full text, and all metadata [163].
- **Batch OCR:** There's also a `batch_extract_text` endpoint that can process multiple files in one call [164] [165]. It limits to 10 files per batch to protect performance [166]. It essentially loops through each file, does the same validation and OCR as single, and collects results [167] [168]. It also tries to do a **consolidated analysis** – combining all texts to detect overall document type and running AI on the combined content [169] [170]. This is brilliant for cases where a "document" spans multiple files (e.g., separate deed pages). The consolidated result can provide a holistic analysis (with fallback error handling if it fails) [171] [172]. The batch response includes each file's result and an optional consolidated analysis [173].

Bad: One major issue: these endpoints are declared async but internally call blocking IO (Vision API via `google.cloud` client, PIL image processing, PyMuPDF). Since they don't use `await` for these calls, they will block the event loop, potentially slowing other requests. This is mitigated a bit by using the background_jobs for heavy tasks (the `jobs.py` endpoint might offload some calls). But for `ocr.py` endpoint as written, calling `vision.ImageAnnotatorClient()` and `.text_detection()` is synchronous (the Google client is not async). This could tie up a UVicorn worker for seconds per page. Similarly, `fitz.open()` and iterating PDF pages is CPU-bound. In high load, this is not scalable. Also, for

large PDFs, reading all pages into memory (`pix.tobytes`) could be memory intensive. Batch endpoint could exacerbate that by doing many in one go.
Another minor: storing OCR results uses a fixed `confidence_score=85` as placeholder since Google's API doesn't give one overall [174] – that's fine as a placeholder, but might be misleading to consumers.

Improvements: Ideally, offload OCR work to the background job system instead of doing in request context. In fact, the project *does* have `batch_ocr.py` and uses `jobs.py` to handle OCR asynchronously for potentially long tasks. Perhaps the UI often calls the `/batch-ocr` or uses the background jobs for real heavy lifting. If not, consider converting the endpoints to simply enqueue a job and return a job ID for client polling. This would greatly improve scalability. For the sync code itself: ensure that Vision API calls time out or handle exceptions (the code does check `response.error.message` and raise HTTPException accordingly [175] [176], which is good). Logging any failures will also help debug. Finally, if performance is a concern, reduce duplication between `ocr.py` and `ai.py` (which has a similar OCR function). There might be overlap that could be refactored into common utility functions to avoid maintaining two versions. But overall, the OCR endpoint is feature-rich and robust in functionality – just consider moving heavy work out of the main thread.

ai.py – AI Analysis Endpoints

Good: The `ai.py` endpoints focus on analysis of text and translation. Key features: - **Structure:** It defines request/response models for DocumentAnalysis and OCR, similar to ocr.py but geared towards text input [177] [178]. This suggests one can send raw text to `/ai/analyze_document` instead of a file. - **Combined pipeline:** Functions like `extract_text_from_image` and `pdf_to_images` are defined here too (similar to ocr.py but slightly simplified to always call Vision synchronously) [179] [180]. The presence of these might be legacy now that ocr.py exists, but they serve the same purpose: get text from images/PDF. - **Main endpoints:** - `POST /ai/extract_text` in ai.py allows providing `file_id` or `path` directly and returns OCRResponse [181] [182]. It duplicates some logic from ocr.py: it resolves file path either from ID (ensuring user owns the file) [183] or directly if path given, checks existence and extension [184], then processes PDF vs image similarly to earlier (with `extract_text_from_image` calls) [185] [186]. This endpoint seems to be a variant of ocr but perhaps meant for quick use. It returns combined text and pages just like ocr.py's version [187]. - `POST /ai/translate` likely to translate given text (not shown in snippet, but presumably calls the translation service). - `POST /ai/analyze_document` would take a chunk of text and return AI-extracted structured data by calling `process_document_with_ai` [188] [189]. If the same logic as used in ocr.py's AI part, it will do translation if needed, document type detection, comprehensive extraction, specific field extraction and utilities extraction, combining all into a detailed JSON [190] [191] [192]. The structure of `DocumentAnalysisResponse` indicates it returns `document_type, extracted_data, general_data, processing_status` [193]. Likely `processing_status` might indicate if fallback was used etc., given `process_document_with_ai` populates such fields [194] [195]. - **Dependency injection & Auth:** All these endpoints use `Depends(get_current_active_user)` so only authenticated users can use AI services [196]. Good for cost control (ensuring only valid users call expensive OpenAI API). - **Use of external libs:** The code properly uses PyMuPDF to get images at 300 DPI for better OCR accuracy [197], and cleans HTML from Google directions by regex to generate route descriptions (in google_maps, not shown here, but similar attention to detail). It's evident the team tried to maximize information quality from AI and OCR.

Bad: There is some duplication between ai.py and ocr.py – both have `extract_text_from_image` and PDF processing functions, and similar endpoints. This could lead to maintenance overhead. Possibly the project evolved and ocr.py superseded some of ai.py's functionality or vice versa. It might be confusing which endpoint to use for OCR: `/ocr/extract_text` or `/ai/extract_text` – they appear to overlap. Also, similar to ocr.py, these functions call blocking Vision API calls in an async context (no `await`), which can block the server thread. The credentials logic here uses `settings.GOOGLE_CLOUD_CREDENTIALS_PATH` whereas ocr.py used `GOOGLE_APPLICATION_CREDENTIALS` environment variable in a different way [198]. Maintaining two separate credential sources (settings vs env var) could cause confusion – ideally unify these.

Improvements: Consider consolidating OCR logic in one place. Perhaps deprecate one of the duplicate endpoints or have `/ocr/...` delegate internally to the logic in ai.py or vice versa. This would reduce code repetition. Additionally, for the heavy AI calls (`process_document_with_ai` can be slow as it hits GPT-4), using background jobs is advisable. The `jobs.py` endpoints (not fully shown) might allow something like submitting a job for cross-validation or batch parsing, which indeed uses `parse_documents` and such via background [199] [200]. For synchronous analyze_document, maybe it's fine for reasonably sized text, but keep an eye on OpenAI API latency. On the credentials, ensure that both ocr.py and ai.py rely on the same config keys to avoid one failing if only the other is set. In summary, reduce redundancy and possibly offload heavy tasks – otherwise, the AI endpoints are well-designed to integrate the intelligence layer into the app.

maps.py – Location & Maps Endpoints

Good: This module likely provides endpoints for geocoding, directions, and places around a property. From what we see: - It sets up routes like `/maps/` or `/location/` (the router is included as `/maps` and `/location` in api.py [201]). These endpoints probably allow the frontend to fetch location details (like reverse geocode coordinates to address, get route descriptions, find nearby amenities, etc.). - The service behind (app/services/google_maps.py) is very thorough, as we saw: - Static map URL generation, - Directions route narrative (which picks out key steps and summarizes distance/duration) [202] [203], - Reverse geocoding to get address components and then enhancing them with Sri Lanka administrative divisions [204] [205], - Distance matrix for multiple origins/destinations [206] [207], - Nearby places search using Google Places, filtering for type and radius and returning a simplified list of name/vicinity/types/rating, etc. [208] [209]. It even does a quick distance calculation for places (though approximate) [210]. - The endpoints likely expose these functions. For example, possibly: - GET `/location/reverse-geocode?lat=x&lng=y` returning the enhanced address (combining Google and local division info). - GET `/maps/directions?origin=X&dest=Y` returning the narrative description (like the one created in `generate_route_description()` [211] [212]). - GET `/maps/places?lat=x&lng=y&type=school` returning nearby places data. - The code ensures graceful failure if API key is not set (the service functions return `{"error": ...}` instead of throwing) [213] [214] – so endpoints can just forward that to users. - **Benefit:** By having these endpoints, the frontend doesn't need to call Google APIs directly (keeping API keys secret on backend and also allowing additional processing like filtering or caching). The local enhancements (like adding nearest city defaults [202] or mapping admin divisions via a `srilanka_admin_divisions` helper [215]) demonstrate localized customization beyond what Google provides out-of-the-box.

Bad: The main risk is that these calls are also synchronous HTTP calls to Google inside FastAPI. E.g., `requests.get(GEOCODING_BASE_URL…)` is used directly [216] [217]. This can slow the response if Google API is slow. Again, an asynchronous approach or pre-fetching data might help. Rate limiting might also be necessary if these endpoints are hammered, to avoid exceeding Google API quotas or incurring high costs (maybe the global RateLimitMiddleware covers it). Also, the `find_nearby_places` code doesn't page through results – Google Places returns at most 20 results and a `next_page_token` if more; the code

doesn't handle multiple pages, which could be a limitation (though in practice 5km radius likely returns manageable results).
Additionally, because these rely on external APIs, error handling is important – the code does check API response status and returns an error field if not OK [218] [219]. That's good, but the endpoints should ensure to propagate these errors properly (likely just returning the dict as JSON).

Improvements: If usage of these features grows, consider making these endpoints asynchronous by using `httpx` with asyncio (or simply document that they may take a second or two to respond due to external calls). Implement caching for repeated queries (could be as simple as in-memory dict caching lat,lng -> address for reverse geocode, or using Google's Place ID as key to avoid multiple of same search). Also, extend support for additional map features if needed (e.g., caching static map images to avoid repeated Google fetch if the same map is requested often). Right now the implementation is solid – just ensure to keep API keys secure and possibly externalize them (the code reads `settings.GOOGLE_MAPS_API_KEY` which presumably comes from env, so that's good [213]).

regulations.py – Regulatory Compliance Endpoints

Good: The regulations endpoint likely serves data about local building regulations, given the `regulation_database.py` service. That service is very detailed: - It defines categories for UDA, Municipal, Urban Council, etc., with applicable areas and document types [220] [221]. - It likely allows queries like "given a location (lat,lng) and property type, what regulations apply?" The function `get_applicable_regulations` does exactly that: 1. determine planning authority via `detect_planning_authority(latitude, longitude)` [222], 2. get location context (perhaps admin divisions) [223], 3. determine applicable categories based on authority and location [224] (using the categories defined above, maybe matching district or area names), 4. build a compliance requirements list (mandatory docs, recommended, stages) using templates for that authority [225]. The result includes coordinates, planning authority info, applicable regulations, and compliance requirements [226]. - This means the endpoint can return a structured set of rules the valuer must consider (e.g., if in Colombo under UDA, it might list mandatory docs like Development Plan approval, etc.). This is a huge value-add for compliance checking. - The service also supports storing regulatory documents (RegulationDocument model) and linking them to locations (RegulationLocationAssociation) [55] [227]. Endpoints might allow retrieving lists of regulation docs relevant to an area or downloading them. The models track document metadata (title, category, type, applicable areas, etc.) and even versioning (superseded_by relationship) [228] [229]. That's a very forward-thinking feature – possibly enabling a library of zoning laws, etc., accessible from the app. - The compliance checklist template model stores things like mandatory_documents, conditional_documents, approval_stages for each regulation category [230] [231]. The `build_compliance_requirements` likely uses these to output what the user needs. In code, they even craft recommendations for final checks (like if warnings count >5, suggest addressing them) [131]. - Overall, this regulatory component is highly commendable – it shows the project isn't just doing AI but also ensuring legal compliance aspects.

Bad: Most of this appears to be implemented at the service layer, but how much is surfaced via endpoints is unclear. Possibly not all features (like uploading regulation docs) have UI or endpoints yet. If not, some code might be currently unused (e.g., the RegulationDocument model might not be actively utilized without an endpoint to query them). Unused code isn't harmful per se, but it could age. Also, the `detect_planning_authority` function is mentioned but we haven't seen its implementation – if it relies on some external data or shapefiles, errors could occur. The code calls it and if it returns an error and no fallback, it just returns that error [232]. That's fine, but ensure that function is robust (if not implemented, this feature may not fully work).

Improvements: If not present, add endpoints like: - `GET /regulations?lat=x&lng=y` to return `get_applicable_regulations` output (likely exists). - Endpoints to list available regulation documents by category/area, and to fetch a specific document (by ID) for reading/downloading. This would leverage the RegulationDocument model. - Possibly an endpoint to run a compliance *checklist* for a given report (though the validation_engine might cover it). For example, after filling data, user could hit "validate compliance" and the backend uses location + property info to ensure all required docs are provided. From a code perspective, maintain the mapping data (e.g., if new categories or authority areas are needed, update the dictionaries). The code currently lists some city names in lowercase arrays (e.g., UDA applicable_areas include "colombo", "gampaha" etc. [233]). That matching is string-based, which can be brittle if input has different casing or spelling – but likely the enhance_location_with_admin_divisions function standardizes names. Just ensure consistency (maybe store all lowercase and lower incoming names). All in all, the regulatory component is a strong differentiator – continuing to refine and integrate it with the rest of the system (UI prompts when something is non-compliant, etc.) will make the app truly professional.

Backend – Service Utilities

document_generation.py – Report PDF/DOCX Generator

Good: This file is very extensive and is crucial – it programmatically generates the valuation report documents. Positive points: - **Use of libraries:** It uses ReportLab for PDFs and python-docx for Word, which are solid choices. The code sets up templates, styles, and flowable elements carefully. For instance, it creates a letterhead with valuer info and contact, reference number and date tables [234] [235], and a centered title and subtitle [236]. The attention to styling (custom ParagraphStyles for headings, letterhead, etc.) means the PDF will look professional (e.g., darkblue text for headings, proper spacing) [237] [238]. - **Dynamic content:** The generator pulls data from the `Report` and related models. E.g., it uses `user.full_name`, title, qualifications, etc., to populate the letterhead and signature areas [239] [240]. It also iterates over `report.data` sections in `_add_report_sections_to_pdf` (not shown fully, but implied) to add each section's content with proper headings [241] [242]. In DOCX generation, it similarly adds headings and paragraphs step by step for each report section (Introduction, Property Identification, Location, etc.) [243] [244]. This shows the code mirrors the wizard structure to output a narrative document. - **Numbers to words:** It imports `convert_lkr_to_currency_words` [245], so likely when filling the valuation summary, it converts numeric values to words (e.g., 1,000,000 to "One Million Rupees") for a formal touch. - **Appendices and photos:** The code likely handles adding images or maps in appendices (since `appendix_generator.py` is imported in reports.py and presumably used to generate map snapshots, etc.). The DocumentGenerationService might incorporate images via `Document.add_picture` for DOCX or by adding flowables in PDF. - **Separation of concerns:** They smartly separated template creation into perhaps a `template_engine` (imported as create_template_engine) [245] for loading any custom Jinja templates, and a `number_to_words` utility. This makes the document service more focused on assembling content rather than formatting every tiny detail – some could be offloaded to templates or separate functions. - **Testing:** A `test_document_generation.py` exists, meaning there are likely tests to ensure PDF/DOCX generation runs without errors given sample data. That's important because these libraries can be finicky; having tests ensures the generation logic doesn't break.

Bad: This file is over 1000 lines [246] [247] – quite complex. Maintaining it might be hard if the report format changes. Minor issues: - **Hardcoded styles and text:** The content like section titles “1. INTRODUCTION”, “For Banking and General Purposes” subtitle [248] [249] are hardcoded. If the requirements change (say a different report structure), one must edit code. Using external template files (e.g., a Word template or JSON for sections) could make it more flexible. However, given the complexity of dynamic content, code may indeed be the most flexible. - **Large in-memory buffers:** Generating PDFs and DOCX in memory (`BytesIO`) and possibly adding images could use a lot of RAM for big reports. The code does this to return a BytesIO [250]. If a report has many photos, the PDF assembly should be watched for performance. (The background job approach helps – these run outside the main request thread). - **Edge cases:** Not sure if every field is handled. E.g., in DOCX generation, they assume `report.property_address` exists for intro [249]. If some optional data is missing, the output might have odd phrasing (they do safe_str checks, which default missing values to "N/A" or blank appropriately [251] [252], which mitigates this). - **No user editing**: The generated report text is entirely automated. If a valuer wants to tweak some wording, there’s no simple way except editing the DOCX after generation. This might be acceptable, but perhaps they considered it – maybe that’s why they output DOCX (editable) and PDF.

Improvements: If possible, break the file into smaller modules or use templates for sections. For instance, having a small Jinja template for the introduction paragraph with placeholders might be easier to adjust than Python string concatenations. However, using code allows complex conditional logic (like listing boundaries or adding bullet points only when certain data present [253] [254]). Another improvement is to incorporate the AI suggestions – e.g., if AI extracted some textual description, merge it into relevant sections (perhaps the code’s `report.data` already includes AI-populated text). Continue testing the output on sample data to ensure formatting is correct (the test likely does that). Also, consider adding page numbers or table of contents if needed for long reports – ReportLab can do that with PageTemplates. All said, the implementation covers the essentials for a polished report, which is a significant achievement.

email_service.py – Email Sending Utility

Good: This service handles sending emails (e.g., delivering the final report to a client). Notables: - It supports plain text and HTML bodies and attachments [255] [256]. It properly constructs a multipart email with both plain and HTML alternatives, which improves email client compatibility [257] [258]. - It reads SMTP settings from config (server, port, username, etc.), and provides an `is_configured()` check to avoid sending if creds aren’t set [259] [260]. This prevents runtime errors – it logs an error and returns False if email is not configured [261]. - It uses TLS (`server.starttls()`) and logs in with credentials before sending [262]. Following this with `server.send_message(msg)` is straightforward and secure (assuming SMTP over TLS). - The `send_report_delivery_email` method is a high-level wrapper that constructs a friendly message to the recipient with an introduction, some notes, and attaches the PDF and DOCX if provided [263] [264]. The HTML version even adds some styling (a highlight box for custom message, etc.) [265] [266]. This yields a professional-looking email without needing an external template. - Attachments are handled in `_add_attachment`, which covers both BytesIO and bytes content, encoding them properly [267] [268]. It sets a generic `application/octet-stream` or uses provided MIME type, and ensures Base64 encoding for safe transmission [269] [268].

Bad: One risk: the SMTP password is held in memory (settings) and used here; ensure it’s kept safe and not logged. The code doesn’t log the password (good), but just be careful when enabling debug logging for SMTP (not done here, so fine). Also, the service currently sends emails synchronously. If SMTP server is slow, this could block for a few seconds. In a web request context, it might be okay for user-initiated “send report” action, but ideally sending could be offloaded to background job too. Another minor thing: it doesn't explicitly close the SMTP connection (context manager `with smtplib.SMTP(...)` ensures closure on exit [262], so that's fine).

Improvements: Possibly integrate with the job system – e.g., when a report is finalized, spawn a background job to send the email so the API responds immediately while email goes out asynchronously. This avoids making the user wait for email sending. Also, consider using environment-specific settings: e.g., in dev, one might not want to actually send emails (could direct to console or a dummy SMTP). The code could allow switching off email sending by leaving SMTP unconfigured (it already does – if not configured it just logs error and returns False [261]). Further, maybe allow templating of the email body (if the user wants to customize the message text or branding). Currently, the text is hardcoded but includes placeholders for custom_message and reference numbers which is good [270] [266]. Overall, the email service is simple and effective for its purpose.

appendix_generator.py – Appendices (Maps/Images) Creation

Good: The name suggests it generates additional content (like maps snapshots, comparable sales tables, etc.) to include as appendices in the report. Though we didn't open it due to time, based on usage: - It's likely used in background jobs for PDF/DOCX generation (`jobs._handle_pdf_generation` calls `document_service.generate_pdf_report(report)` which might internally call appendix_generator to embed things like location maps or photos). - If it generates collated images (like combining photos or stamping them), that's a plus. Possibly it uses `google_maps.generate_static_map_url` and then fetches the map image to include in the report appendix. - This separation means the main document generation can focus on text while appendices (visuals) are handled here, which is a neat design.

Bad: Without specifics, hard to critique. One guess: if it calls external APIs (Google Static Maps) to retrieve images during PDF generation, that could slow down generation or fail if offline. Caching maps by coordinates might be wise to avoid repeated calls.

Improvements: If not implemented, consider making appendices optional – e.g., only generate heavy maps if user wants them, to save time. And utilize caching for external image fetches to reduce API usage/cost.

advanced_logger.py – Enhanced Logging & Analytics

Good: This utility augments the default logging with structured logs and performance metrics: - It sets up multiple handlers: console, file, JSON log file, error-specific file [271] [272]. This means logs are not only seen on console but also saved (e.g., `valuerpro_detailed.log`, `valuerpro_analytics.jsonl`, `valuerpro_errors.log`) [273] [274]. The JSONL logs allow later analysis of events (AI usage, validation results, etc.) in a structured way – great for debugging and analytics. - It provides convenience methods to log specific events: `log_ai_extraction_attempt/success/failure` record AI usage with model, text length, time taken, etc. [275] [276] [277]; `log_data_validation_results` logs how many errors/warnings after validation [278]; `log_cache_performance` for cache hits/misses timing [279]; `log_user_workflow_step` to record when a user completes a wizard step (with potentially large `step_data` size noted but not fully stored) [280]; `log_system_performance_alert` for any thresholds exceeded [281]. These specialized logs indicate the team planned to monitor system usage and performance proactively. - The `get_performance_summary()` aggregates metrics collected (like average extraction time per doc type, error rates per doc type, total extractions) [282] [283]. This is used by ExpertConfig to auto-tune settings if needed [284] [285] – an innovative feedback loop: e.g., if AI extraction is slow, ExpertConfig reduces batch size or increases timeouts [286]. - It gracefully handles cases where advanced logging isn't available by a debug message (the ExpertConfig catches ImportError for advanced_logger and skips optimization if not present) [287].

Bad: Overengineering caution – while this is great for a production system at scale, it adds complexity. If not actively monitored, these logs could grow large (the JSON log especially). The code does not indicate a log rotation – writing to `*.log` files will eventually need rotation to prevent disk fill. Also, the performance metrics (like self.performance_metrics dict) aren't persisted beyond runtime – on restart, they reset. That's fine if only used for adaptive tuning in a long-running server, but if restart is frequent, the data might be sparse. Another potential issue: writing to files in a container (if deployed via Docker) could be lost or fill the container FS; often logs are better sent to STDOUT or a logging system. They do log to console as well, so maybe that covers it.

Improvements: Implement log rotation (perhaps using Python's TimedRotatingFileHandler or external logrotate in deployment) to manage file sizes. Also, ensure thread-safety if these methods are called from multiple threads (the code appends to lists in performance_metrics without a lock; in practice, since logging calls are usually thread-safe at the handler level, it might be okay, but direct list mutation might need care). If the adaptive config is indeed used, monitor that it doesn't thrash (changing settings too often). Given the complexity, decide which metrics are truly needed and consider simplifying if not all are used – e.g., if `log_user_workflow_step` is not being consumed anywhere, it might be overkill to log potentially large step data JSON for every step. However, having these hooks is beneficial for future analytics (like to see where users drop off in the wizard). Overall, advanced_logger.py is a sign of an enterprise-level outlook on the product – maintain it by integrating with a monitoring solution (perhaps parse the JSON logs to create dashboards of average OCR time, etc., which would be very cool).

expert_config.py – Dynamic Configuration Manager

Good: This component tries to auto-optimize settings and enable advanced features dynamically. For example: - It defines dataclasses for various config sections (AIExtractionConfig, PerformanceConfig, ValidationConfig, LoggingConfig) with sensible defaults (like GPT-4 model, token limit, temperature, caching enabled, etc. in AI config) [288] [289], performance thresholds (memory, slow op threshold 5s, etc.) [290], validation toggles for strictness, and logging preferences (enable JSON logs, analytics) [291] [292]. - On initialization, it loads any existing JSON config files from a `config/` directory for each section, so configurations can persist across runs if changed [293] [294]. If no file, it saves the defaults to file for easy editing next time [295]. This is a nice touch – it means an admin could tweak a JSON in the `config` folder to change system behavior without changing code. It creates the config directory if not present [296]. - The manager's `update_config` method allows changing specific config values at runtime and immediately saves them to file [297] [298] – so if the system detects something and wants to adjust (or via an API endpoint for admin), it's straightforward. - It attempts an `_setup_auto_optimization()` which uses the ExpertLogger's performance summary [299]. If performance stats exist, it calls `_optimize_based_on_performance` to adjust settings. For example, if average extraction time > threshold, it reduces batch size or increases timeout, and if things are fast, it might plan to increase batch or so (though code for speeding up isn't shown, presumably more logic would be added) [300] [301]. This is forward-thinking – an auto-tuning system to keep things efficient without manual intervention. - Essentially, ExpertConfig sets the stage for a system that learns and adapts – rare in many applications and quite impressive.

Bad: This is mostly infrastructure for future use; it's not clear how actively it is used currently. Potential downsides: - If mis-tuned, auto-changes could be counterproductive (e.g., if an outlier spike temporarily reduces batch size unnecessarily). Without a lot of data, it's tricky to auto-tune reliably. There's also no persistence of performance stats beyond what ExpertLogger keeps in-memory. - Complexity: This subsystem may confuse developers who aren't aware of it – "why did batch_size suddenly change?" if it logs "Updated ai_extraction.batch_size" [302] after auto-optimization. It does log updates and warnings for unknown params, which helps audit changes. - The dependency on `advanced_logger.expert_logger` means if advanced_logger isn't imported, auto-optimization is skipped (they catch ImportError) [287]. That's fine – it fails safe (just logs debug and does nothing) [303]. But if advanced_logger is loaded later or not at all, this might never run, making ExpertConfig moot. - Storing configs as JSON is fine, but concurrent writes (if multiple update events triggered quickly) could race – however, updates likely occur infrequently and on a single process, so not an issue in practice.

Improvements: If the product finds a need for runtime tuning, flesh out this system: e.g., schedule `_setup_auto_optimization` to run periodically (currently it runs once at startup only). Could integrate with a cron or background thread to periodically adjust based on recent metrics (though be careful in async frameworks). Also, more optimization rules could be implemented – currently, only extraction time influences batch size and timeout [304]; perhaps error rates could disable caching or switch models if GPT-4 fails (just hypothetical). If this proves unnecessary, it could be left as is or removed to reduce complexity. At minimum, ensure any config updates are communicated – perhaps surface current config in an admin UI for transparency. Since it's not critical path, leaving it dormant doesn't harm the application, but it's a powerful tool waiting to be used.

Frontend – Next.js Application

Next.js Structure & Global Config

Good: The project uses Next.js App Router (Next 13+), with TypeScript and Tailwind – modern choices. The `next.config.ts` and other config files were present (jest, eslint, postcss config) which shows proper tooling setup. The code organization parallels the backend structure: e.g., `frontend/src/lib/api.ts` for API calls, `frontend/src/components/wizard` for the multi-step form, `frontend/src/app` for pages (login, register, dashboard, etc.). This logical separation makes it easy to maintain.

Bad: One must ensure the Next.js version (14 per README) and App Router features are stable – Next 13/14 had some changes, but presumably it works. Tailwind usage likely ensures responsive design (as seen in classnames with grid, px, etc., in page.tsx). Nothing glaringly bad in structure.

Improvements: Keep Next.js up to date and monitor any needed changes for new versions of the App Router, but this is just general maintenance. Possibly split some config (like environment-specific API URLs are handled via `.env.local` as README suggests) – indeed, they use `NEXT_PUBLIC_API_URL` for pointing frontend to backend [305], which is good.

Authentication Pages (Login & Register)

Good: The login page uses React state and a custom hook for form validation (`useFormValidation` presumably from `useAsyncState` or `useFieldValidation`) [306] [307]. It validates email format and password via imported helpers (validateEmail/Password) on blur, providing immediate feedback by setting error messages in state [308] [309]. This gives users a smooth experience. On submit, it calls `login()` from `useAuth` (which likely wraps `authAPI.login`) and, on success, redirects to `/dashboard` [310]. It properly handles loading state to disable the button and shows a general error message on failure (using `parseAPIError` to get a friendly message) [311] [312]. The UI is accessible (labels with `sr-only`, error texts, etc.) and mobile-responsive classes from Tailwind are used for inputs and container.

The register page collects extensive info (full name, experience, etc.) which matches backend's UserCreate fields [313] [314]. It live-validates the password (showing requirements) and ensures password confirmation matches [315] [316]. On submit, it also calls `authAPI.register`, and on success redirects to login with a message query param indicating success [317] [318]. This is a good flow – user sees a notice to login after registering. The form is split into Basic Info and Professional Info sections with clear headings and helper text [319] [320], improving usability. Optional fields (registration_no, etc.) are not marked required, and the code doesn't require them (`ValuerProfileCreate` allows None) so it's fine.

Bad: Minor UI nitpicks: The register form is long; on small screens it may be a lot of scrolling. They do use a grid for some fields (e.g., password and confirm side by side on sm screens) [321] [322], which helps. Validation: email is required but they don't appear to validate its format on client (they rely on input type=email and backend). That's acceptable.

Improvements: Perhaps break the registration into multi-step if user feedback suggests it's overwhelming (though since all fields are likely needed by compliance, one form is fine). After registration redirect, they pass a message query – the login page could read that and show a success alert "Registration successful, please login." If not implemented, adding that would improve UX. Also, any additional validation can be added to password (the UI text suggests requirements "8+ chars, A-Z, a-z, 0-9, special" [323], but the actual `validatePassword` function should enforce those – presumably it does on client or server).

useAuth & API integration

Good: The `useAuth` hook (not fully shown) probably provides context for authentication (storing the token, user info, and `login/logout`). The `authAPI.login` function in `api.ts` uses FormData to call `/auth/login` and on success stores tokens in localStorage/sessionStorage [324] [325]. Storing access_token in localStorage is common, though not the most secure if XSS – but since they plan secure cookies according to README, maybe they'll adjust to set token as cookie from server (the backend doesn't currently, it returns token JSON). They do clear tokens and redirect on 401 globally via httpClient interceptor [326] [327]. That's very good – it handles session expiration by forcing re-login. The `authAPI.register` and profile updates map neatly to backend endpoints [328] [329], making the front-back integration clean and predictable.

Bad: localStorage token storage has some risk (XSS could steal tokens). A more secure approach is to have backend set an HttpOnly cookie on login. But implementing that would require adjustments. The code as is must ensure to **clear** tokens on logout (likely useAuth handles that). Another edge: after login, they don't immediately fetch user profile – but `authAPI.getMe` exists [330]. Possibly `useAuth` calls `getCurrentUser()` on mount or so to populate context. If not, some slight delay in showing user info might occur. Not critical.

Improvements: If concerned about security, consider switching to cookie-based auth (the backend has SecurityMiddleware which might handle secure cookies, but not sure). If sticking to SPA token auth, ensure all API calls include the token. The `httpClient` likely attaches `Authorization: Bearer <token>` header automatically (the code doesn't show it, but maybe `httpClient` is configured to do so by reading localStorage token). This should be confirmed – otherwise, APIs wouldn't authenticate. Setting up `httpClient` interceptors to inject the token from storage is a common practice (perhaps done in http-client.ts). So, in summary, just maintain the auth flow and consider security enhancements as needed.

WizardProvider & WizardLayout (15-Step Wizard State Management)

Good: This is the heart of the frontend. The WizardProvider holds the entire form state across steps and provides functions to navigate and save. Great points: - **Centralized State:** It uses `useReducer` to manage a `WizardState` with currentStep index, a nested `data: ReportWizardData` object for all steps, a `reportId` for the draft report, loading status, errors, and a dirty flag [331] [332]. Initial state sets all sections (reportInfo, identification, location, etc.) to empty objects or defaults, matching backend schema structure [333] [334]. This mirror of backend data shape ensures easy mapping when sending to API. - **Reducer Actions:** Actions handle setting step index, updating data (either whole data or specific step sub-object) [335] [336], setting reportId, loading, errors, etc. This is clean and prevents unnecessary re-renders (only changes cause relevant state updates). - **Context:** The provider exposes methods like `nextStep`, `previousStep`, `goToStep`, `updateStepData`, etc., which encapsulate the logic of moving through the wizard [337] [338]. For example, `canGoToStep(step)` likely checks that all prior steps have at least minimal data before allowing jumping ahead – ensuring sequential completion (in the code snippet, `canGoToStep` isn't shown, but likely uses `getStepCompletion()` which returns an array of which steps are completed) [339]. This prevents users from skipping steps unless prior ones are done, a good UX pattern. - **Auto-save:** The provider sets up an effect to auto-save after 2 seconds of inactivity whenever state is dirty and a reportId exists [340] [341]. This means as the user fills the wizard, the data is periodically sent to the server (calling `saveReport()` which calls backend to update the report and valuation summary) [342] [343]. This reduces risk of data loss and allows multi-device or stop-resume usage. The auto-save uses a timeout and clears it on cleanup to prevent multiple rapid calls [344]. - **API Integration:** The `saveReport` function does a backend PUT for basic report info (ref, purpose, dates, etc.) [345] and then if a valuation summary exists in state, posts that via `reportsAPI.createValuationSummary` [346]. `saveStepData` handles specific steps like identification (calls `reportsAPI.createProperty` with identification data) and valuation lines [135] [133]. This granular saving is efficient – it doesn't always send the whole huge payload, only what's needed. Also, it means the backend endpoints can be simpler (one endpoint per resource). - **Form Validation & Completion:** The `validateStep(step)` function checks required fields in that step and returns an array of errors [347] [348]. It has helper `isEmpty()` to detect missing values robustly (handles undefined, empty string, NaN, empty array) [349]. It likely covers each step's important fields: e.g., ensures identification has lot and plan number, location has district/province, etc., numeric fields are valid using `validateNumber()` provided inside [348]. This ensures the user cannot proceed if something critical is missing or malformatted (there might be custom logic per step within that function, not fully visible). The use of `getStepCompletion()` presumably uses similar logic to mark completed steps for the progress UI [350]. - **UI/UX:** WizardLayout component renders the top progress bar with step circles and titles, highlighting current and completed steps with colors/icons [351] [352]. It's scrollable (for 15 steps on small screens) and clickable: clicking a completed step circle triggers `onStepClick(index)` to jump backwards or to already-done sections [351] [353]. Navigation buttons are appropriately enabled/disabled (`Previous` disabled on first step, `Next` disabled if cannot go next) [354] [355]. The "Save & Continue" button is present (on Next or final step) – perhaps on final step it might say "Save & Finish" if configured (they pass a prop `saveLabel` which defaults to "Save & Continue" but could be changed) [356]. - **Polish:** Warnings like marking slow requests in backend and providing process time hint that could tie into UI if needed, but the UI probably doesn't expose that. However, the front shows errors (e.g., if an API call fails, how are errors surfaced? The state has an `errors` field which presumably could be used to display field-level errors if validation fails on backend or saving fails) [357] [358]. For instance, if backend returns validation issues on save, the provider might set state.errors and the UI for that step could show them next to fields. This error handling path isn't fully shown, but the structure supports it.

Bad: The wizard logic is complex – a bug in it could derail the user experience. For example, if `validateStep` misses a scenario, a user might proceed with incomplete data and cause backend errors later. Also, the interplay between auto-save and manual navigation needs to be tested: if a user quickly clicks "Next" before auto-save triggers, does the data still get saved? Likely yes, because moving next could call `saveStepData` for that step (the provider's `nextStep` doesn't explicitly call save, but maybe the Next button's onClick triggers a save via `onSave` prop to WizardLayout [359] which calls `saveReport()` or similar). Actually, the `WizardLayout` has a Save button separate from Next, which is interesting – perhaps they allow saving at any time or saving partial progress (maybe `onSave` is used for an explicit save action, whereas normally moving next doesn't require clicking save because of auto-save). This could confuse some users ("Do I need to hit Save or just Next?"). It might have been simpler to save on Next automatically and label it "Next (autosaved)". But this is a design choice.

Improvements: Continue rigorous testing of the wizard – simulate all steps, refresh mid-way to ensure auto-saved data reloads properly (the provider does `loadReport(reportId)` on mount if in editMode [360], presumably fetching existing data from backend and dispatching SET_DATA). Also, refine step validation messages to be user-friendly (like highlight missing fields). The UI currently might just prevent navigation without clearly saying why. Perhaps add inline messages or highlight required fields in red if `validateStep` fails (the provider's `validateStep` returns errors but the code doesn't show usage – likely the UI for each step calls it on submit and displays an alert or marks fields). Using the `errors` state, they could map field errors to specific inputs. Implementing that feedback loop will improve UX (the backend also returns validation details in 422 errors, but front-end validation should catch most). Finally, consider performance: 15 steps with lots of state might be fine on modern devices, but if any step has heavy components (like a map or large file upload preview), code-splitting or dynamic loading might be needed. Next.js App Router automatically code-splits pages, so each step's component likely loads only when needed, which is good. Just ensure memory usage doesn't balloon if all data (like images) is kept in state – maybe images are just stored as file references on server rather than in the React state.

UI Components – Wizard Steps & Others

Good: Each step component (IdentificationStep, UtilitiesStep, etc.) likely corresponds to a section of the form. E.g., IdentificationStep will have inputs for Lot No, Plan No, etc., and use `useWizard()` to get `state.data.identification` and `updateStepData`. They probably update WizardProvider as the user types (which would mark state dirty and eventually auto-save). This is efficient as all data is in one place. The context avoids prop drilling across 15 pages. Also, context ensures consistency – e.g., the Navigation can highlight completed steps via `getStepCompletion` and WizardProvider knows which steps have data.

Bad: Not much known without looking at them, but one general caution is to avoid too many re-renders. If the entire form state is one object, updating one field might re-render all fields unless steps are isolated. Using `updateStepData` which shallow merges only one step's sub-object helps limit changes to that step's portion of state [336]. Possibly each step component can be wrapped in `React.memo` to avoid re-render if unrelated state changes (like another step's data). If performance issues arise (15 steps isn't too high, but each has multiple fields), consider such optimizations.

Improvements: Standard form enhancements – e.g., adding **auto-suggest** or lookups in relevant steps (maybe they did: e.g., LocationStep might integrate Google Places auto-complete for address or use the Maps endpoints for reverse geocode). Integrating the maps features into the wizard (like showing a map preview when coordinates are set, or a "Use my location" button that calls the reverse_geocode) would improve usability. Given the comprehensive backend, the UI could surface these nicely (if not already implemented). For example, an **EnvironmentalStep** might call the regulations API to warn if property is in a landslide zone (if NBRO category applies). The structure is there; it's about connecting the dots on the frontend for a richer, interactive form.

Conclusion

Overall Assessment: This codebase is impressively comprehensive and well-architected. **Good practices abound** – from rigorous input validation and security checks on the backend [155] [32], to a responsive and state-managed frontend wizard that auto-saves user progress [340] [354]. The integration of AI (OCR and GPT-4 analysis) is done carefully with error handling and fallback logic [361] [195], and the system's design anticipates real-world needs like performance monitoring, scaling via background tasks, and regulatory compliance.

Each file generally has far more **strengths** than weaknesses: - The backend code is modular, making it easier to maintain (each concern in its own endpoint or service file). It is also defensive – checking preconditions and catching exceptions to return meaningful HTTP errors rather than crashing [142] [362]. - The few **issues** noted (blocking calls in async endpoints, some duplicated logic, and the complexity that might challenge new contributors) are not fundamental flaws but areas to watch as the project grows. - The **improvements** suggested largely revolve around performance optimizations (using async or background jobs for heavy tasks), security enhancements (token storage and usage), and maintainability (reducing duplicate code, keeping documentation in sync).

Given the project is preparing for production, it's in **very good shape**. The code reflects an "expert developer" mindset: security middleware to prevent common vulnerabilities, exhaustive validation to prevent bad data, logging for observability, and extensibility to adapt to future needs. By addressing the few bad points (especially ensuring asynchronous handling of long processes and fine-tuning the user experience in the wizard), the team can significantly improve reliability and scalability.

In summary, ValuerPro's code is robust and thoughtfully engineered. With continued attention to the identified areas, it will be well-prepared for a stable production release, providing both a powerful feature set and a maintainable codebase for the long term.

---

1 2 3 4 5 6 README.md
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/README.md

7 8 9 10 Makefile
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/Makefile

11 12 CLAUDE.md
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/CLAUDE.md

13 14 15 16 17 18 19 20 21 305 ci-cd.yml
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/.github/workflows/ci-cd.yml

22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 main.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/main.py

37 38 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 362 ocr.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/api/api_v1/endpoints/ocr.py

39 40 41 42 43 44 45 48 49 50 51 52 53 54 55 56 57 58 59 227 228 229 230 231 models.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/models.py

46 47 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 136 reports.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/api/api_v1/endpoints/reports.py

60 61 62 63 64 67 schemas.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/schemas.py

65 66 68 69 70 71 72 73 74 validation.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/core/validation.py

75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 199 200 background_jobs.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/core/background_jobs.py

95 96 97 201 api.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/api/api_v1/api.py

98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 auth.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/api/api_v1/endpoints/auth.py

115 324 325 326 327 328 329 330 api.ts
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/frontend/src/lib/api.ts

133 134 135 137 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 357 358 360 WizardProvider.tsx
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/frontend/src/components/wizard/WizardProvider.tsx

177 178 179 180 181 182 183 184 185 186 187 193 196 197 198 ai.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/api/api_v1/endpoints/ai.py

188 189 190 191 192 194 195 361 ai_extraction.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/services/ai_extraction.py

202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 google_maps.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/services/google_maps.py

220 221 222 223 224 225 226 232 233 regulation_database.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/services/regulation_database.py

234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 document_generation.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/services/document_generation.py

255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 email_service.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/services/email_service.py

271 272 273 274 275 276 277 278 279 280 281 282 283 advanced_logger.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/utils/advanced_logger.py

284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 expert_config.py
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/backend/app/utils/expert_config.py

306 307 308 309 310 311 312 page.tsx
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/frontend/src/app/auth/login/page.tsx

313 314 315 316 317 318 319 320 321 322 323 page.tsx
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/frontend/src/app/auth/register/page.tsx

350 351 352 353 354 355 356 359 WizardLayout.tsx
https://github.com/Malith-nethsiri/valuerpro-project/blob/f371aecec5eeed5329ca87c2f4f39632cc9f61db/frontend/src/components/wizard/WizardLayout.tsx